

**Faculty of Engineering and Technology Electrical and Computer Engineering   
Computer Science Department**

**Comp438 – QA course**

**Title: Course project**

|  |  |
| --- | --- |
| *Students Name* | *Student Number* |
| *Anas Al Sayed* | ***1221020*** |
| *Abd Al-Raheem Yaseen* | ***1220783*** |
| *Rakan Omar* | ***1221334*** |
| *Ahmad Rimawi* | ***1220343*** |
| *Mohammad Nemer* | ***1222300*** |

**Instructor**: **Dr**. Faisal Shehadeh

**Date: 15/5/2025**

1. **Table of Contents**

[**II.** **Table of Figures** 4](#_Toc200962658)

[**III.** **List of Tables** 4](#_Toc200962659)

[**1.** **Project Overview** 5](#_Toc200962660)

[**1.1 Application description** 5](#_Toc200962661)

[**1.2 Requirement Analysis** 5](#_Toc200962662)

[**1.2.1 Functional Requirements (FR)** 5](#_Toc200962663)

[**1.2.2 Non-Functional Requirements (NFR)** 6](#_Toc200962664)

[**1.3 Scope of Testing** 7](#_Toc200962665)

[**1.4 Member Responsibilities** 7](#_Toc200962666)

[**2.Test Plan Document** 8](#_Toc200962667)

[**2.3 Test Schedule** 8](#_Toc200962668)

[**2.4Test Strategy** 9](#_Toc200962669)

[**2.3 Tools Used** 9](#_Toc200962670)

[**2.4 Risk Analysis** 9](#_Toc200962671)

[**2.5 Traceability Matrix** 10](#_Toc200962672)

[**2.6 Test Plan Approvals** 10](#_Toc200962673)

[**3.1 Static Code Inspection** 11](#_Toc200962674)

[**3.1.2 TaskEditViewModel.kt** 14](#_Toc200962675)

[**3.1.3 DateTime** 19](#_Toc200962676)

[**3.1.4 MainActivityViewModel.kt** 22](#_Toc200962677)

[**3.1.2 TaskListfregment.kt** 24](#_Toc200962678)

[**3.1.5 TaskListfregment.kt** 27](#_Toc200962679)

[**3.2 Black-box test case designs** 31](#_Toc200962680)

[**1-Add task with valid data.** 31](#_Toc200962681)

[**3.2.2 Abd Al-Raheem Yaseen - 1220783** 38](#_Toc200962682)

[BTC06 : 39](#_Toc200962683)

[BTC07 : 40](#_Toc200962684)

[BTC08 : 41](#_Toc200962685)

[**3.2.3 Rakan Omar – 1221334** 42](#_Toc200962686)

[1- Create Task with Valid Date 42](#_Toc200962687)

[2- Create Task with Invalid Leap Day 43](#_Toc200962688)

[3- Create Task with Invalid Leap Day 44](#_Toc200962689)

[4- Create Task with Invalid Day (Day = 00) 45](#_Toc200962690)

[**3.2.4 Ahmad Rimawi – 1220343** 47](#_Toc200962691)

[Test Case 1: Add a New Task 47](#_Toc200962692)

[Test Case 2: Mark Task as Complete 48](#_Toc200962693)

[Test Case 3: Filter by Completed Tasks 49](#_Toc200962694)

[Test Case 4: Edit an Existing Task 50](#_Toc200962695)

[**3.3 white-box test case designs** 55](#_Toc200962696)

[**2-** hasDueTime()**.** 56](#_Toc200962697)

[**3-** setDueDateAdjustingHideUntil(long newDueDate)**.** 57](#_Toc200962698)

[**4-** isNotifyAfterDeadline()**.** 58](#_Toc200962699)

[3.3.2 Abd Al-Raheem Yaseen - 1220783 59](#_Toc200962700)

[3.3.3 Rakan Omar – 1221334 64](#_Toc200962701)

[**1-** Constructor and getters**.** 64](#_Toc200962702)

[**2-** testPlusAndMinus: 65](#_Toc200962703)

[**3**- testStartEndOfDay: 66](#_Toc200962704)

[**4**- testStartEndOfDay: 67](#_Toc200962705)

[**5**- testTimezoneConversion: 68](#_Toc200962706)

[6- testEqualsAndHashCode: 69](#_Toc200962707)

[III.3.4 Ahmad Rimawi – 1220343 70](#_Toc200962708)

[Test Case 1: Set Task and Ensure State is Updated Internally 70](#_Toc200962709)

[Test Case 2: Search Query Updates Menu Query in State 70](#_Toc200962710)

[Test Case 3: Drawer Close Clears Query Field 71](#_Toc200962711)

[Test Case 4: Set Filter Updates State Internally 71](#_Toc200962712)

[**3.4 Automated test scripts using (Katalon)** 73](#_Toc200962713)

[**3.5 JMeter test plans and result analysis** 75](#_Toc200962714)

[**3.5.2 Screenshots and Configuration Details** 76](#_Toc200962715)

[**3.5.3Observations** 80](#_Toc200962716)

[**3.5.4 Analysis & Conclusion** 80](#_Toc200962717)

[**3.6 Jira issues and test management reports** 81](#_Toc200962718)

[**3.6.1 Jira Configuration** 81](#_Toc200962719)

[**3.6.2 Backlog Management** 81](#_Toc200962720)

[**3.6.3 Test Case Coverage & Jira Reports** 84](#_Toc200962721)

[**3.6.4 Integration Highlights** 84](#_Toc200962722)

[**3.6.5 Conclusion** 84](#_Toc200962723)

[**3.7 Regression Testing** 84](#_Toc200962724)

[**4.** **Summary of Activities and Improvements** 85](#_Toc200962725)

# **Table of Figures**

[Figure 1:Add task with valid data](about:blank) 24

[Figure 2:Task Shown in list after added successfully](about:blank) 24

# **List of Tables**

[Figure 1:Add task with valid data](about:blank) 24

[Figure 2:Task Shown in list after added successfully](about:blank) 24

[Figure 4: The New Task Name is appearance in the list](about:blank) 26

[Figure3: Replace title with "Updated Title"](about:blank) 26

[Figure 6:every 15 minutes](about:blank) 28

[Figure 5:Select when overdue](about:blank) 28

[Figure 7:Tap checkbox to mark as complete](about:blank) 29

[**Figure 8: testIsCompleted().** 43](#_heading=h.8czfvt9ptl3m)

[Figure 9:testHasDueTime(). 44](#_heading=h.mt1ru34hxsyh)

[**3- setDueDateAdjustingHideUntil(long newDueDate).** 45](#_heading=h.qdrd60i7anj1)

[Figure 10: tsetSetDueDateAdjustingHideUntil(). 45](#_heading=h.o13swl4wbvvl)

[Figure 11:testIsNotifyAfterDeadline() 46](#_heading=h.bhvufmmbuiji)

# **Project Overview**

**1.1 Application description**  
**Title**: QA Testing and Automation for the Tasks Android App  
**Source**: <https://github.com/tasks/tasks> **Platform**: Android (Kotlin and Java)  
**Justification**: The Tasks app is a widely used open-source productivity app. It is written in Java, follows MVVM architecture, supports CalDAV sync, and includes task management features that make it suitable for functional, structural, performance, and regression testing.

## **1.2 Requirement Analysis**

### **1.2.1 Functional Requirements (FR)**

**FR01:**The user shall be able to create a new task with a title, due date, priority, and description.

**FR02:**The user shall be able to edit an existing task's details including title, due date, recurrence, and notes.

**FR03:**The app shall allow users to mark tasks as completed and restore them if needed.

**FR04:**The app shall automatically save task updates or allow users to manually save changes.

**FR05:**The app shall support adding subtasks under a parent task for hierarchical task tracking.

**FR06:**The user shall be able to attach files or photos to a task.

**FR07:**The app shall provide the ability to assign tags and location-based reminders to tasks.

**FR08:**The app shall allow users to set reminders (alarms) for task start time, due date, and overdue deadlines.

**FR09:**The app shall support repeating tasks with customizable recurrence rules (e.g., daily, weekly).

**FR10:**The user shall be able to delete, duplicate, and move tasks between lists or folders.

**FR11:**The app shall display task notifications based on the user's selected reminder settings.

**FR12:**The app shall allow syncing of tasks with external calendar services like Google Calendar and CalDAV.

**FR13:**The app shall support offline task editing and sync changes once the device is reconnected to the internet.

### **1.2.2 Non-Functional Requirements (NFR)**

**Platform & Compatibility**

**NFR01**. The system shall support devices running Android 8.0 (API 26) or higher.

**NFR02**. The app shall be developed using Kotlin and Java with SDK 21 and Android Architecture Components (MVVM).

**NFR03**. The app shall support data persistence using Room Database.

**Performance & Responsiveness**

**NFR04**. The system should launch the main screen within 2 seconds under normal conditions.

**NFR5**. Notifications and reminders shall be delivered reliably and punctually according to user-defined preferences.

**Offline & Reliability**

**NFR06**. The system shall provide full offline functionality for all core features.

**NFR07**. The system should ensure reminders are delivered reliably, even in background mode.

**NFR08**. The app shall recover gracefully from crashes, saving unsaved data where possible.

**User Interface & Accessibility**

**NFR09**. The system shall provide both dark theme and light theme for accessibility.

**NFR10**. The system should support user interface scaling for different screen sizes.

**Error Handling & Fail Safety**

**NFR11**. The system should fail gracefully with user-friendly error messages.

## **1.3 Scope of Testing**

Modules to Test: Task creation, editing, deletion, reminders, filtering, settings.  
Testing Types Applied: Static testing, black-box testing, white-box testing, regression testing, automation testing, performance testing.

## **1.4 Member Responsibilities**

|  |  |  |
| --- | --- | --- |
| Group Task | Assigned Leads | Description |
| Requirement Analysis | **Anas (Lead)**, all contribute | Analyze features (tasks, reminders, lists, sync, themes). Document functional + non-functional requirements. |
| Performance Testing with JMeter | **Mohammad (Lead)**, supported by Anas | Identify testable endpoints (if any) or simulate I/O-heavy flows like reminders, task creation in bulk. |
| Regression Testing Plan | **Abd (Lead)** | Identify previous test cases affected after making changes (e.g., task update logic). Show understanding via test reruns. |
| Test Management (Jira) | **Ahmad (Lead)** | Create test cases, track defects, map coverage. All members upload 4 black/white-box test cases + 3 automation scripts. |
| Test Plan, Traceability, Risk Docs | **Mohammad (Lead),** supported by Anas | Collect and document traceability matrix, risk analysis, and test strategies. Include tools used (JUnit, Appium, JMeter, Jira). |
| Final Presentation + Summary | **Rakan (Lead)** | Format slides, include lessons learned, demo QA tools used. Each member presents their testing class and results. |

**2.Test Plan Document  
  
  
2.1 Objective:** The purpose of this test plan is to ensure the quality and reliability of the mobile app developed using Java and Kotlin. We will verify all major functionalities such as creating tasks, editing, and deleting tasks.

**Scope:** Our testing covered task creation, editing, deletion, reminders, filtering, and settings. We applied static, black-box, white-box, regression, automation, and performance testing types

**2.2 Test Strategy**

* Static testing on classes**.**
* White-box testing using JUnit for logic methods**.**
* Black-box testing via UI: creation/edit/delete tasks**.**
* Regression testing after code changes **.**
* Automation testing using Katalon for core flows**.**
* Performance testing using JMeter**.**

**2.2 Test Objectives**

**Functional Testing:** Ensure users can create new tasks, edit them, and delete them**.**

**Performance Testing:** Ensure the app performs well when managing multiple tasks**.**

## **2.3 Test Schedule**

|  |  |
| --- | --- |
| Phase | Date |
| Requirement Analysis | May 10–13 |
| Static Testing | May 14 |
| Unit Testing (JUnit) | May 15–17 |
| Automation Testing | June 2-15 |
| JMeter Load Testing | May 18–21 |
| Regression Testing | May 21–24 |
| Final Test Plan + Slides | May 25–28 |

### **2.4 Test Strategy**

We will use a combination of manual and automated testing:

**Unit Testing**: We’ll use JUnit to test small units of code.

**UI Testing**: We’ll use Katalon for automating UI interaction tests.

**Performance Testing**: We’ll use JMeter to test the app’s performance under load.

## **2.3 Tools Used**

|  |  |
| --- | --- |
| Tool | Purpose |
| JUnit | White-box unit tests |
| Kaatalon | UI automation tests |
| JMeter | For performance testing |
| Jira | Test case and defect tracking |
| Android Studio | Coding/debugging |

## **2.4 Risk Analysis**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Risk ID | Description | Likelihood | Impact | Solution |
| R1 | Task logic fails in regression | Medium | High | Retest affected modules |
| R2 | Notification delivery bug | High | High | Add Katalon test coverage |
| R3 | Jira not mapped to FRs | Low | Medium | Use traceability matrix |
| R4 | Emulator instability | Medium | Low | Use physical devices if needed |

## **2.5 Traceability Matrix**

|  |  |  |  |
| --- | --- | --- | --- |
| Req ID | Requirement | Test Case IDs | Type |
| FR01 | Add task | TC-BB-01, TC-WB-03 | Black, White |
| FR02 | Edit task | TC-BB-05 | Black |
| FR03 | Mark task complete | TC-BB-04, TC-WB-01 | Black, White |
| FR08 | Add reminders | TC-BB-03, TC-WB-04 | Black, White |
| NFR01 | API compatibility | Device testing | Compatibility |
| NFR04 | Load screen in 2s | JMeter run | Performance |
| NFR11 | Error handling | Katalon scripts | Functional |

### **2.6 Test Plan Approvals**

The test plan needs approval from Project **Manager** (Mohammad nemer) and **QA Manager** (Abdalraheem yassen ) before starting the execution.

**3.Test Artifacts**

**3.1 Static Code Inspection  
  
3.1.1 Task.kt**

**TASK QA – Static Code Inspection Report  
Reviewer:** Anas Al Sayed **Class Under Review:** Task.kt **Module:** org.tasks.data.entity.Task **Review Date:** 17/5/2025

1. **Class Overview:**  
   Task is the core entity class representing a user task in the **Tasks Android application**. It is annotated with Room’s @Entity for database persistence and includes serialization support via @Serializable.   
   This class encapsulates task properties **(**title, due date, notes, etc.**)**, status flags (completed, deleted), reminder configuration, recurrence, and sync metadata.   
   It also provides utility functions for equality comparison, reminder checks, and transient state management.
2. **Responsibilities**  
   • Define task data structure for persistence and serialization  
   • Track task status (completed, deleted, recurring, read-only, etc.)  
   • Manage reminder and sync metadata using transitory flags  
   • Provide helper methods for comparison (e.g., up-to-date checks)  
   • Encapsulate task recurrence and date logic  
   • Enable local-only runtime state (transitoryData) for UI behavior or sync control
3. **Inspected Methods**

**A.  isCompleted / isDeleted**  
• Simple checks based on timestamp fields  
• Efficient and readable  
• No issues identified

**B.  insignificantChange(task)**  
• Performs a detailed comparison of nearly all task fields  
• Used for change detection before syncing  
• Could be refactored with a TaskDelta or data diff utility to reduce repetition

**C. hasDueTime(dueDate: Long)**  
• Checks if due time (not just date) exists by checking modulus  
• Valid approach, clearly named and intuitive  
• Consider improving readability with documentation or a Duration wrapper

**D. suppressSync(), suppressRefresh(), isSuppressRefresh()**  
• Manages transient sync suppression state  
• Uses synchronized block and HashMap for flags  
• Might benefit from replacing raw string keys with enums/sealed classes

**E. googleTaskUpToDate(), caldavUpToDate(), microsoftUpToDate()**  
• Perform equality checks tailored to different sync targets  
• Some redundancy exists across methods — DRY principle could improve maintainability  
• High coupling with external sync structure

**4. Inspected Features**

|  |  |
| --- | --- |
| **Feature** | **Notes** |
| **Field Definitions** | **Covers all expected task metadata (title, priority, dates, etc.)** |
| **Room Annotations** | **Uses @Entity, @PrimaryKey, @ColumnInfo correctly** |
| **Transient Fields** | **Uses transient HashMap for runtime-only data (good practice)** |
| **Custom Logic** | **hasDueTime(), isCompleted(), isRecurring(), equalsNullable(), etc.** |
| **Reminder Flags** | **Bitmask flags implemented with constants** |

**5. Observed Issues & Suggestions**

|  |  |  |
| --- | --- | --- |
| **Issue** | **Description** | **Recommendation** |
| **public fields** | **Many fields are public (e.g., title, dueDate). This breaks encapsulation.** | **Change fields to private + add getters/setters** |
| **equals/hashCode missing** | **The class defines insignificantChange(), but does not override equals() or hashCode().** | **Consider adding overrides for equality comparison in lists/adapters** |
| **Lack of Javadoc** | **Most methods lack Javadoc-style comments** | **Add Javadoc to public methods for maintainability** |
| **Suppress flag strings** | **Constants like "suppress-refresh" are repeated as strings** | **Extract constants where missing; minimize hardcoded literals** |
| **No validation on setters** | **setUuid(), setRandomReminder(), etc., do not validate input** | **Consider adding input checks if accessed externally** |
| **isDeleted() implementation** | **Simply checks if deletionDate > 0** | **Consider using a boolean field or better logic for soft deletes** |
| **repeatFrom usage** | **repeatFrom enum logic lacks runtime enforcement** | **Add logic to validate values passed (e.g., if outside enum range)** |

• Extract sync-related comparison logic into a TaskComparator or TaskDiff class  
• Replace transitory string keys with a sealed class or enum for better safety  
• Add a proper hashCode() override to match equals-like logic  
• Reduce coupling by separating reminder and sync utility logic  
• Modularize repetitive comparison code (e.g., extract into reusable private equalsCoreFields method)

**6. Conclusion**  
The Task class is foundational to the Tasks app and effectively models the state and metadata of a user’s task. However, its responsibilities extend beyond data representation — it handles behavioral logic like sync state detection and reminder configuration. While this centralization is understandable, it increases cognitive load and reduces testability.

### **3.1.2 TaskEditViewModel.kt**

**TASK QA – Static Code Inspection Report  
Reviewer:** Abd Al-Rahem -Yaseen  **Class Under Review:** TaskEditViewModel.kt **Module:** org.tasks.ui **Review Date:** 17/5/2025

1. **Class Overview:**  
   TaskEditViewModel is the ViewModel responsible for managing task creation and editing state in the Tasks app. It connects the TaskEditFragment (UI) with data persistence and business logic, including alarms, tags, subtasks, calendar sync, and location data.
2. **Responsibilities**

* Maintain task state (title, description, due date, alarms, etc.)
* Handle user inputs from the UI and update internal state
* Save task to the database using TaskDao
* Schedule and synchronize reminders with AlarmService
* Sync with external services like Google Calendar and CalDAV
* Manage subtasks, tags, locations, and attachments

1. **Inspected Methods**

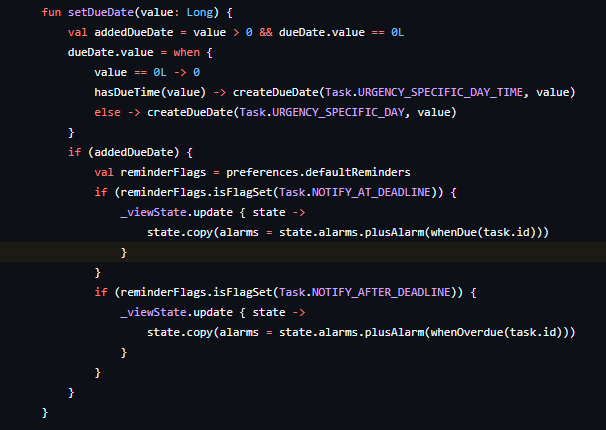
**A. save()**

* Too long and complex (>120 lines), violates Single Responsibility Principle
* Handles too many concerns: saving task, subtasks, tags, alarms, location, attachments
* Should be split into helper methods like saveAlarms(), saveTags(), saveSubtasks()
* Risk: high cognitive load, hard to test, error-prone



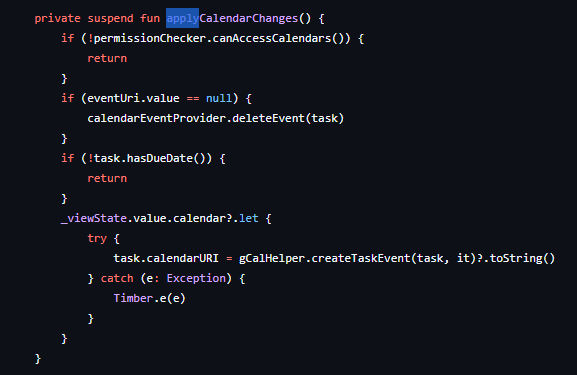
**B. setDueDate()**

* Updates due date and conditionally adds alarms
* Handles flag-based logic using preferences
* Contains duplicate alarm-adding logic also found in setStartDate()



**C. applyCalendarChanges()**

* Handles sync with Google Calendar
* Uses try-catch but does not notify UI on sync failure
* Risk: silent failure may lead to data inconsistency



**D. hasChanges()**

* Determines whether the task has been modified
* Important for deciding when to trigger save
* Contains repetitive comparison logic – could be cleaner with a helper data class

1. **Risks and Smells Identified**

|  |  |  |
| --- | --- | --- |
| Area | Issue | Recommendation |
| save() | Too long, violates SRP | Split into smaller private functions |
| Alarm logic | Duplicated across multiple methods | Centralize alarm addition in one method |
| applyCalendarChanges() | Catches errors but does not notify | Return result or log failure to UI/logcat |
| Preferences & Flags | Hardcoded boolean logic for ring modes | Use enums or sealed classes for better clarity |
| ViewState updates | Lots of .update calls with mutable flows | Risk of race conditions, test for concurrency |

1. **Suggestions for Refactoring**

* Modularize large logic blocks within save()
* Introduce AlarmManager helper class to manage alarm logic
* Use data class TaskDelta to encapsulate changes in hasChanges()
* Improve null safety on task notes, picture URIs, etc.
* Apply separation of concerns: isolate sync logic from ViewModel

1. **Conclusion**  
   TaskEditViewModel is central to the task editing functionality but has accumulated too much responsibility. It handles critical workflows, including task persistence, alarm handling, and external sync. Refactoring and better separation of concerns would improve testability, readability, and maintainability.

### **3.1.3 DateTime**

**TASK QA – Static Code Inspection Report  
Reviewer:** Rakan Hasan Omar **Class Under Review: DateTime**.kt **Module:** org.tasks.time.DateTime.kt **Review Date:** 18/5/2025

1. **Class Overview:**  
   The DateTime class is a utility or model class used to represent and manipulate date and time values throughout the Tasks app. It encapsulates date/time-related logic such as formatting, parsing, timezone handling, and arithmetic operations like adding or comparing dates. This class serves as a core component for date handling in task creation, scheduling, and reminders.
2. **Responsibilities**

* Store and represent date and time values
* Provide utility methods to parse and format dates
* Handle time zone conversions and local/UTC time differences
* Support date-time arithmetic (e.g., adding days, checking expiration)
* Validate date input formats
* Interact with system or app-level time settings
* Serve as an abstraction over native date/time types (e.g., ZonedDateTime, Calendar, or
* Instant)

1. **Inspected Methods**

**A.  Constructor and getters:**

Methods: DateTime(), getYear(), getMonthOfYear(), getDayOfMonth(), getHourOfDay(), getMinuteOfHour(), getSecondOfMinute(), getMillis(), getTimeZone().

* Creates a new GregorianCalendar on every getter call — inefficient.
* No input validation for date/time parameters.
* Stores millis and time zone but recalculates fields repeatedly.
* Fields not immutable; risks accidental modification.
* Lacks useful overrides (toString(), equals()).
* Could cache calendar instance to improve performance.

**B.  Plus, and Minus**

Methods: plusHours(), plusDays(), minusDays(), minusSeconds()

* Creates a new GregorianCalendar on every getter call — inefficient.
* No input validation for date/time parameters.
* Stores millis and timeZone but recalculates fields repeatedly.
* Fields are mutable, risks accidental modification.
* Lacks caching of Calendar instance; performance could improve by caching.
* Could make fields immutable (val) to avoid accidental changes.

**C. Start and End:**

Methods: startOfDay () and endOfDay()

* Clear, intuitive methods returning new DateTime instances (immutability respected).
* Each method creates a new instance, no side effects on shared Calendar.
* Could reduce duplication by extracting common helper function for boundary calculations.
* Overall efficient and easy to understand.

**D. After and Before**

Methods: isAfter() and isBefore()

* Simple, efficient comparisons using stored millis field.
* Clear intent, easy to read and maintain.
* Assumes non-null input, no null checks present (could be added for safety).
* Relies on consistent millis representation across DateTime instances.

**E. Time zone:**

Methods: toUTC(), toLocal() and toTimezone()

* Convenient methods for time zone conversions.
* toTimeZone() avoids creating new objects if timeZone matches efficiently.
* Correctly converts time zone by creating new Calendar instance with same instant.
* Clear, concise, and reusable private helper method for conversions.

**5. Inspected Features**

|  |  |  |
| --- | --- | --- |
| **Area** | **Issue** | **Recommendation** |
| Constructors | No input validation for date/time parameters | Add validation to prevent invalid dates |
| Getters | Recreates “GregorianCalendar” on every getter call; inefficient | Cache or memorize calendar instance |
| Plus/Minus methods | No input validation: “minusDays” calls “plusDays(-days)”, may confuse | Add validation; clarify negative input use |
| startOfDay()/endOfDay() | Code duplication for setting time fields; mutates shared Calendar | Extract helper method; use defensive copy |
| isAfter/isBefore | No null checks on input parameter | Add null validation |
| Timezone conversions | Depends on “getCalendar()” correctness; no caching of conversions | Ensure safe calendar; consider caching |
| equals()/hashCode() | Fields mutable risking inconsistent equality/hashCode; only compares Millis and timeZone | Declare fields “final”; update if new fields added |

1. **Conclusion**  
   The Task class is foundational to the Tasks app and effectively models the state and metadata of a user’s task. However, its responsibilities extend beyond data representation — it handles behavioral logic like sync state detection and reminder configuration. While this centralization is understandable, it increases cognitive load and reduces testability.

### **3.1.4 MainActivityViewModel.kt**

**TASK QA – Static Code Inspection Report  
Reviewer:** *Ahmad Rimawi* **Class Under Review:** MainActivityViewModel.kt **Module:** org.tasks.time.MainActivityViewModel.kt  
**Review Date:** 16/5/2025

1. **Class Overview:**  
   The MainActivityViewModel is a Hilt-injected ViewModel for managing the UI state of MainActivity. It uses Kotlin StateFlow for state management and combines LiveData-like reactive streams with background operations on filters, drawer state, task details, and account data.
2. **Code Inspection Checklist**

**Readability**

* Use of data class State encapsulates UI-related fields well.
* Use of private MutableStateFlow and public asStateFlow exposes only immutable state.
* Function names are self-explanatory (e.g., updateFilters(), resetFilter(), toggleCollapsed()).
* Code could benefit from more comments, especially for non-obvious logic in updateFilters().

**Maintainability**

* Good separation of concerns: filter logic, drawer management, and state updating are compartmentalized.
* Constant values (e.g., magic numbers like 1000 ms in throttleLatest) should be extracted into named constants.
* Use of kotlinx.collections.immutable ensures predictable UI updates.

**Logic and Functionality**

* Redundant filtering check in setFilter:  
  if (filter == \_state.value.filter && task == null) avoids unnecessary work.
* Proper lifecycle management: refreshReceiver is unregistered in onCleared().
* toggleCollapsed() has potential for missed updates if broadcast fails silently.

**Error Handling**

* Exception handling inside taskDao.count(): try-catch logs error via Timber.
* Default values provided for filter when it’s not found via savedStateHandle.
* Error indicator from NavigationDrawerSubheader (hasError) is respected in DrawerItem.Header.

**Performance & Efficiency**

* onStart triggers updateFilters only once at initial launch, which is good.
* updateFilters is throttled using throttleLatest(1000) to avoid rapid UI updates.
* runBlocking for filter loading from savedStateHandle is discouraged in ViewModel constructor. Should use suspend initialization or lazy loading.

**Security**

* No obvious security vulnerabilities. Data access and state management occur via well-defined interfaces.

**Testability**

* ViewModel logic is well-structured for unit testing, especially for:
  + setFilter()
  + queryMenu()
  + toggleCollapsed()
* Some I/O operations (like caldavDao.count()) could be abstracted further to allow mocking.

1. **Suggestions for Improvement**

* Replace runBlocking calls in constructor with coroutine-based initialization using init or a suspend loader.
* Add KDoc comments to complex methods like updateFilters() and toggleCollapsed().
* Extract magic numbers (e.g., 1000 ms in throttleLatest) to a constant val THROTTLE\_DURATION\_MS = 1000.
* Consider exposing critical filter logic for dedicated unit testing (e.g., filter matching, task count fallback).

### **3.1.2 TaskListfregment.kt**

**TASK QA – Static Code Inspection Report  
Reviewer:** Mohammad nemer  **Class Under Review: TaskListfregment**.kt **Module:** org.tasks.ui.TaskListfregment.kt **Review Date:** 17/5/2025

1. Class Overview:  
   The TaskListFragment class in this Android application is responsible for managing the display and interaction with a list of tasks within a to-do list or task management app. It includes functionality for displaying tasks based on various filters, handling task-related actions such as creating, editing, and deleting tasks, and interacting with task data stored in a local database. The fragment also integrates with several UI components, including a toolbar, search functionality, and a floating action button for adding new tasks. Additionally, it supports synchronization with external services to keep the task list up-to-date.  
   This class also manages user interactions through various UI elements, including swipe gestures, toolbar actions, and context menus. It handles both simple interactions like task selection and deletion, as well as more advanced actions such as sorting, filtering, and managing recurring tasks. The fragment makes use of various lifecycle-aware components like ViewModel and LiveData to ensure smooth state management across different screen orientations or configuration changes. Additionally, it integrates with external services like speech recognition for voice-based task creation, and it provides support for managing task categories, priorities, and due dates.
2. **Responsibilities**

• Define task data structure for persistence and serialization  
• Sync task list with external services.  
• Handle UI interactions like search and task actions.  
• Use ViewModel and LiveData for task state.  
• Enable sorting, filtering, and voice commands.

1. **Inspected Methods  
   1.** **onCreateView()**

**2. onMenuItemClick()**

**3. onActivityResult()**

* **What it does:  
  Sets up the screen when first opened.**
* **How it works:**
  + **Prepares the task list (RecyclerView).**
  + **Sets colors based on the current filter.**
  + **Adds pull-to-refresh feature.**
  + **Shows warning banners if needed.**
* **Why it's important:  
  Everything you see when opening the app comes from here!**

* **What it does:  
  Handles all button clicks in menus.**
* **Examples:**
  + **Settings button → Opens settings.**
  + **Voice button → Starts voice input.**
  + **Sort button → Changes task order.**
  + **Clear button → Deletes done tasks.**
* **Special feature:  
  Shows "Are you sure?" popup before deleting.**

**3. onActivityResult()**

* **What it does:  
  Gets results from other screens.**
* **Two main cases:**
  + **Voice input → Makes a new task from speech.**
  + **Tag selection → Adds tags to tasks.**

1. **Simple version:  
   It listens when other screens send data back.**

### **3.1.5 TaskListfregment.kt**

**TASK QA – Static Code Inspection Report  
Reviewer:** Mohammad nemer  **Class Under Review: TaskListfregment**.kt **Module:** org.tasks.ui.TaskListfregment.kt **Review Date:** 17/5/2025

**1. Class Overview**

The TaskListFragment class in this Android application is responsible for managing the display and interaction with a list of tasks within a to-do list or task management app. It includes functionality for displaying tasks based on various filters, handling task-related actions such as creating, editing, and deleting tasks, and interacting with task data stored in a local database. The fragment also integrates with several UI components, including a toolbar, search functionality, and a floating action button for adding new tasks. Additionally, it supports synchronization with external services to keep the task list up-to-date.  
This class also manages user interactions through various UI elements, including swipe gestures, toolbar actions, and context menus. It handles both simple interactions like task selection and deletion, as well as more advanced actions such as sorting, filtering, and managing recurring tasks. The fragment makes use of various lifecycle-aware components like ViewModel and LiveData to ensure smooth state management across different screen orientations or configuration changes. Additionally, it integrates with external services like speech recognition for voice-based task creation, and it provides support for managing task categories, priorities, and due dates.

**2. Responsibilities**

* Define task data structure for persistence and serialization
* Sync task list with external services
* Handle UI interactions like search and task actions
* Use ViewModel and LiveData for task state
* Enable sorting, filtering, and voice commands

**3. Inspected Methods**

**1. onCreateView()**

**What it does:**  
Sets up the screen when first opened.

**How it works:**

* Prepares the task list (RecyclerView)
* Sets colors based on the current filter
* Adds pull-to-refresh feature
* Shows warning banners if needed

**Why it's important:**  
Everything you see when opening the app comes from here!

**2. onMenuItemClick()**

**What it does:**  
Handles all button clicks in menus

**Examples:**

* Settings button → Opens settings
* Voice button → Starts voice input
* Sort button → Changes task order
* Clear button → Deletes done tasks

**Special feature:**  
Shows "Are you sure?" popup before deleting

**3. onActivityResult()**

**What it does:**  
Gets results from other screens

**Two main cases:**

* Voice input → Makes a new task from speech
* Tag selection → Adds tags to tasks

**Simple version:**  
It listens when other screens send data back

**4. Inspected Features**

|  |  |
| --- | --- |
| Feature | Notes |
| UI Components | Uses RecyclerView, SwipeRefreshLayout, BottomAppBar, FAB |
| View Models | TaskListViewModel for task data, MainActivityViewModel for state |
| Dependency Injection | Uses Hilt (@AndroidEntryPoint, @Inject annotations) |
| Task Management | Create/complete/delete/duplicate tasks with coroutines |

**5. Observed Issues & Suggestions**

**1. Class Size and Complexity**

**Issue:**  
The class is extremely large (over 1000 lines) with high complexity.

**Impact:**  
Makes maintenance difficult and violates the Single Responsibility Principle.

**Recommendation:**

* Break down the class into smaller, focused classes (e.g., separate classes for action mode handling, search functionality, banner management).
* Consider using composition over inheritance.

**2. Error Handling**

**Location:**  
Insufficient error handling in coroutines.

**Impact:**  
Silent failures possible.

**Recommendation:**

* Add proper try-catch blocks around critical operations.
* Consider adding error reporting for failures.

**Example:**

lifecycleScope.launch {

    try {

        val tasks = withContext(Dispatchers.IO) {

            taskDao.fetch(data?.getSerializableExtra() as? ArrayList<Long> ?: return@launch)

        }

        val modified = tagDataDao.applyTags(

            tasks.filterNot { it.readOnly },

            // ... other params ...

        )

        taskDao.touch(modified)

    } catch (e: Exception) {

        Timber.e(e, "Failed to apply tags")

        makeSnackbar(R.string.error\_applying\_tags)?.show()

    }

}

**Problem:**  
If the database operation fails, the error is silently swallowed.

**3. Thread Management Issue**

**Location:**  
Frequent database operations executed on the main thread (e.g., taskDao.fetch()).

**Impact:**  
Risk of blocking the main thread, leading to UI freezes or lag.

**Recommendation:**

* Ensure all database operations are moved off the main thread.
* Use coroutines with explicit dispatchers such as Dispatchers.IO to handle database interactions safely in background threads.

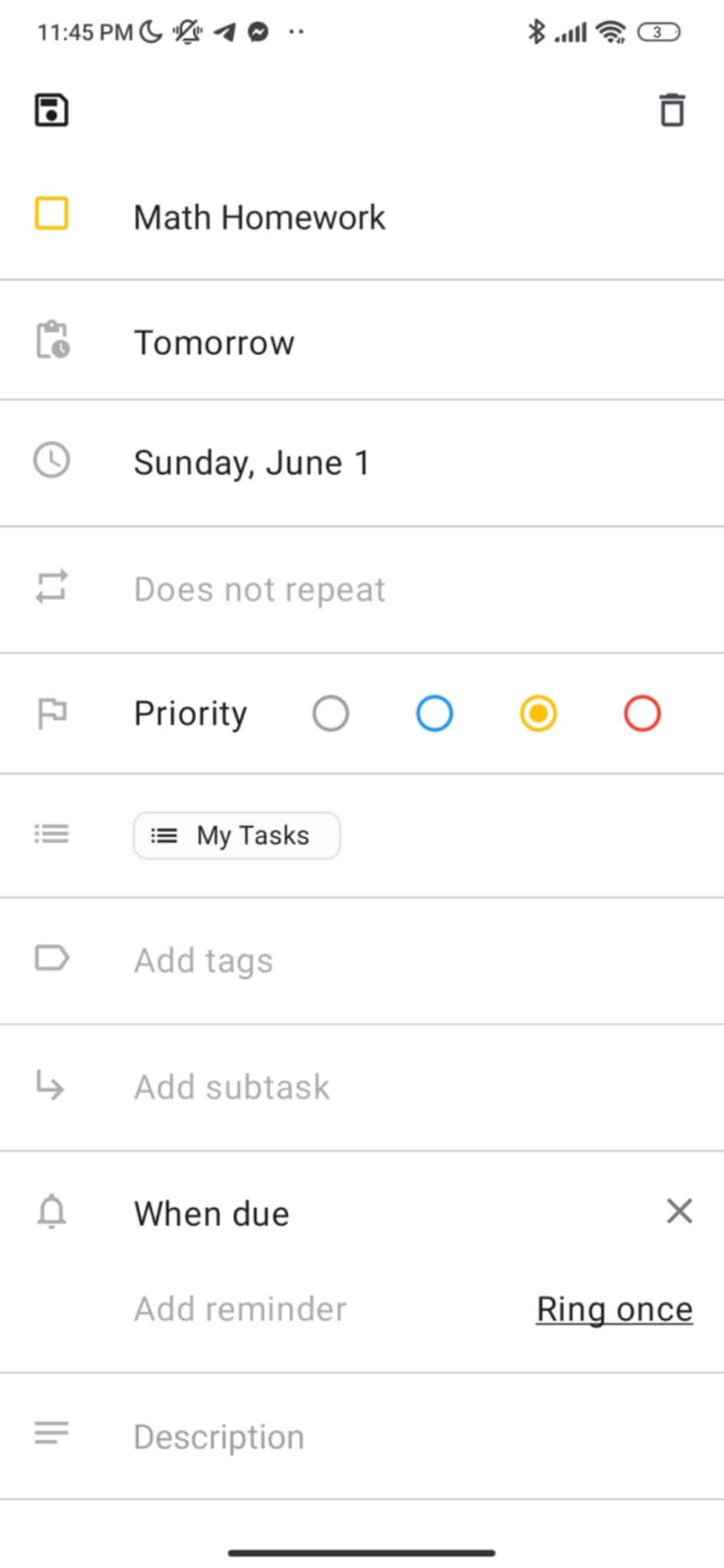
**6. Conclusion**

The TaskListFragment class is large and complex, making it difficult to maintain. It lacks proper error handling in asynchronous tasks and performs database operations on the main thread, risking UI performance issues. To improve, the class should be refactored into smaller, focused components, with better error handling and database operations moved to background threads.

**3.2 Black-box test case designs  
  
3.2.1 Anas Al Sayed - 1221020**

### **1-Add task with valid data.**

|  |  |
| --- | --- |
| Field | Details |
| Test Case ID | BTC01 |
| Title | Add task with valid data |
| Description | |  | | --- | |  |   Verify that the application successfully saves a task with valid input |
| Precondition | User is on "New Task" screen |
| Test Data | Task Name: "Math Homework", Start Date: "2025-05-17", Due Date: "2025-06-01" |
| Test Steps | |  | | --- | | 1. Tap on ➕ to open Add Task screen  2. Enter "Math Homework" in title field  3. Select start date "2025-05-17"  4. Select due date "2025-06-01"  5. Tap on save | |
| Expected Result | Task is saved successfully and appears in task list |
| Actual Result | Task is real saved successfully and appears in task list |
| Priority | High |
| Status | Passed |

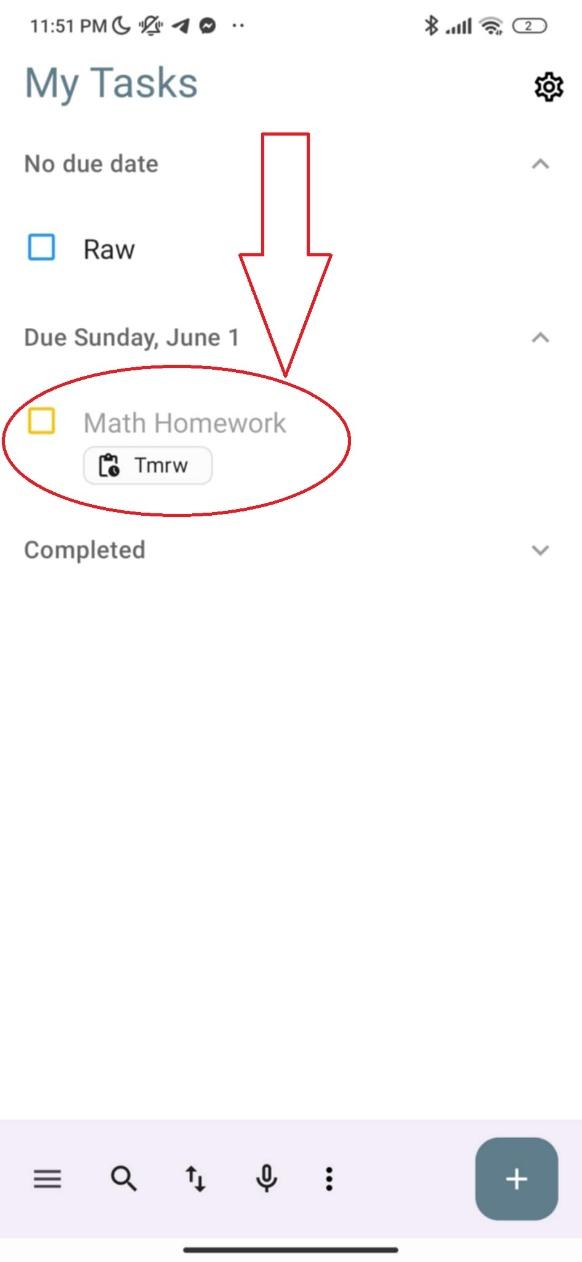
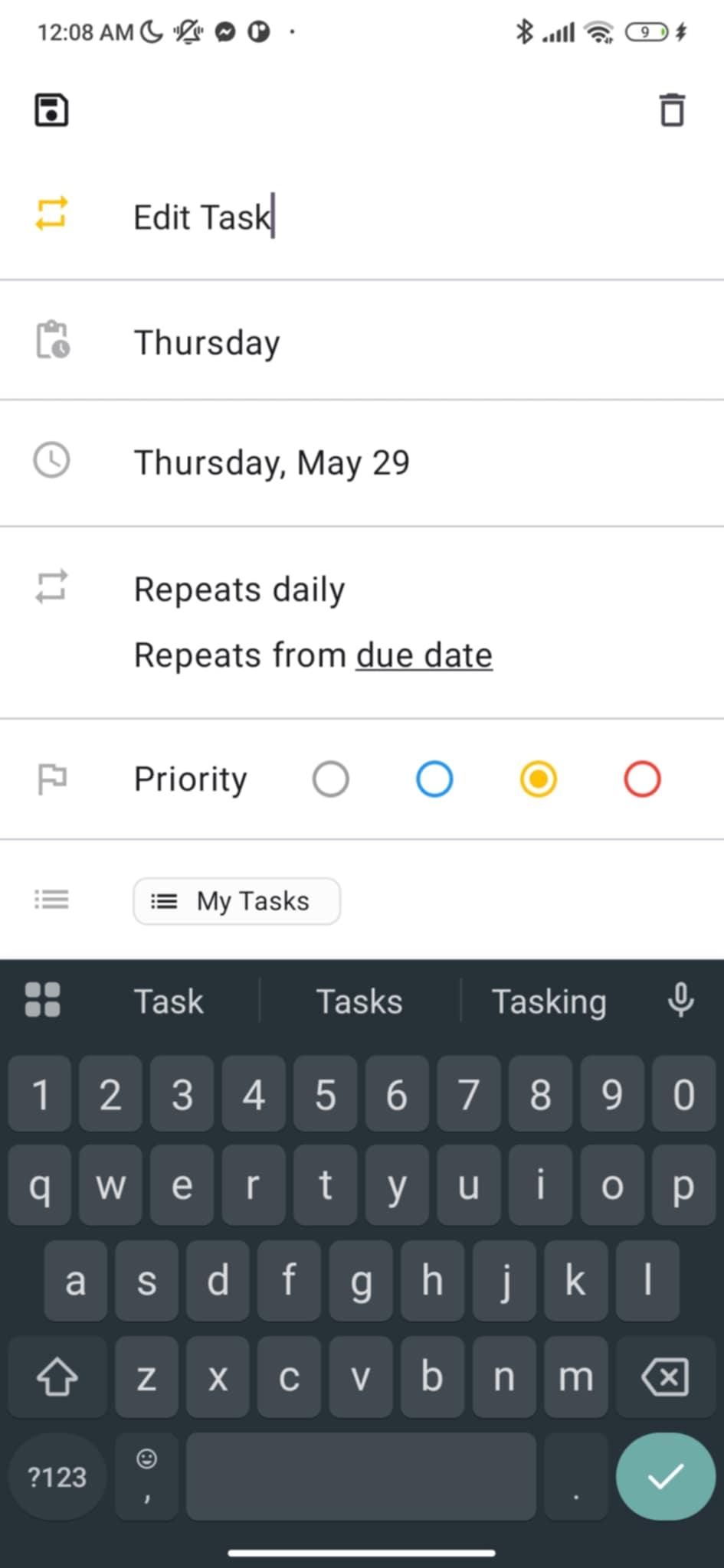


*Figure SEQ Figure \\* ARABIC 1:Add task with valid data*

*Figure SEQ Figure \\* ARABIC 2:Task Shown in list after added successfully*

2- Edit existing task Name

|  |  |
| --- | --- |
| Field | Details |
| Test Case ID | BTC02 |
| Title | Edit existing task title |
| Description | Verify that editing the title of a task updates the task correctly |
| Precondition | A task named "Old Title" exists in the task list |
| Test Data | Original Title: "Old Title", New Title: "Updated Title" |
| Test Steps | 1. Tap on task named "Old Title" to open edit screen  2. Replace title with "Updated Title" 3. Tap Save 4.The New Task Name is appearance in the list |
| Expected Result | Task is updated with the new title and appears as "Updated Title" in task list |
| Actual Result | Task title is successfully updated in the list |
| Priority | High |
| Status | Passed |

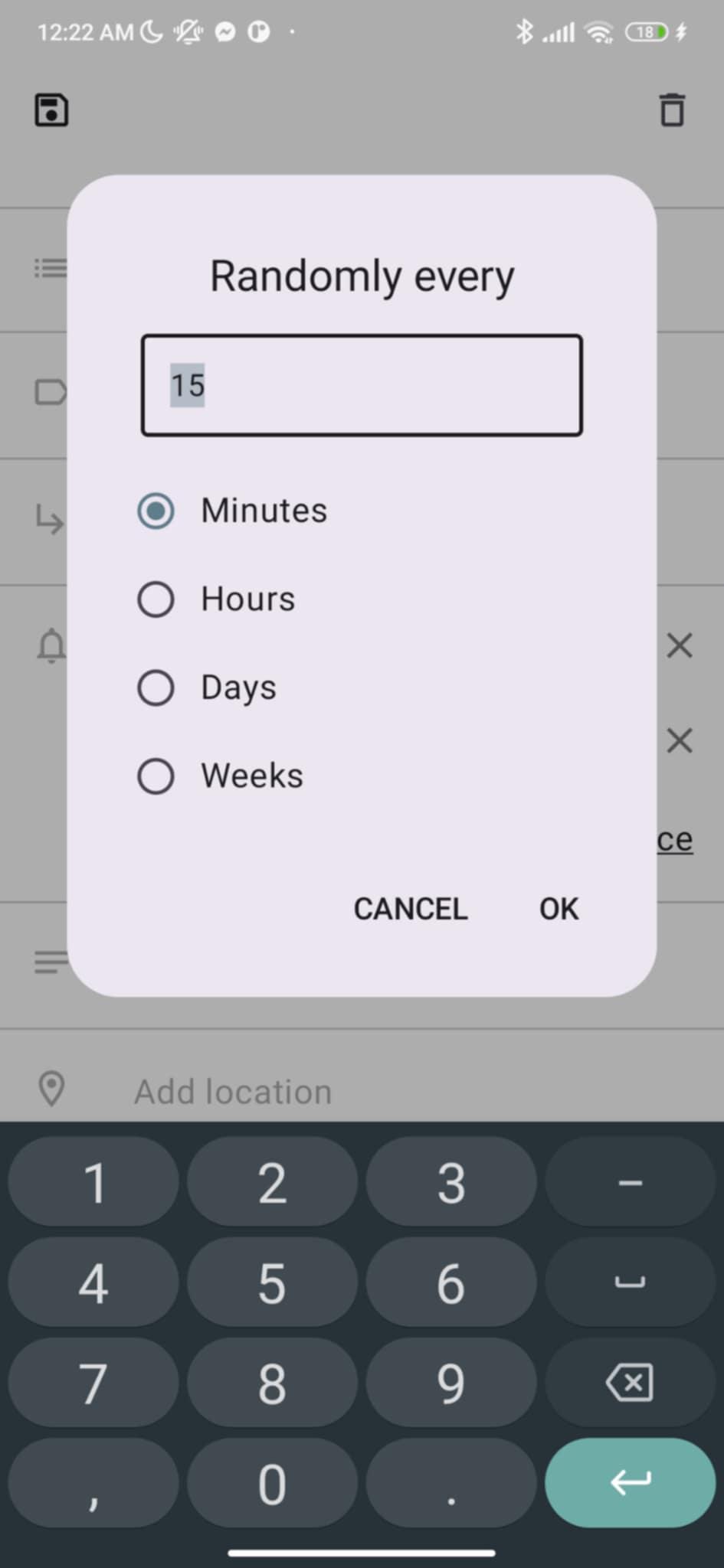
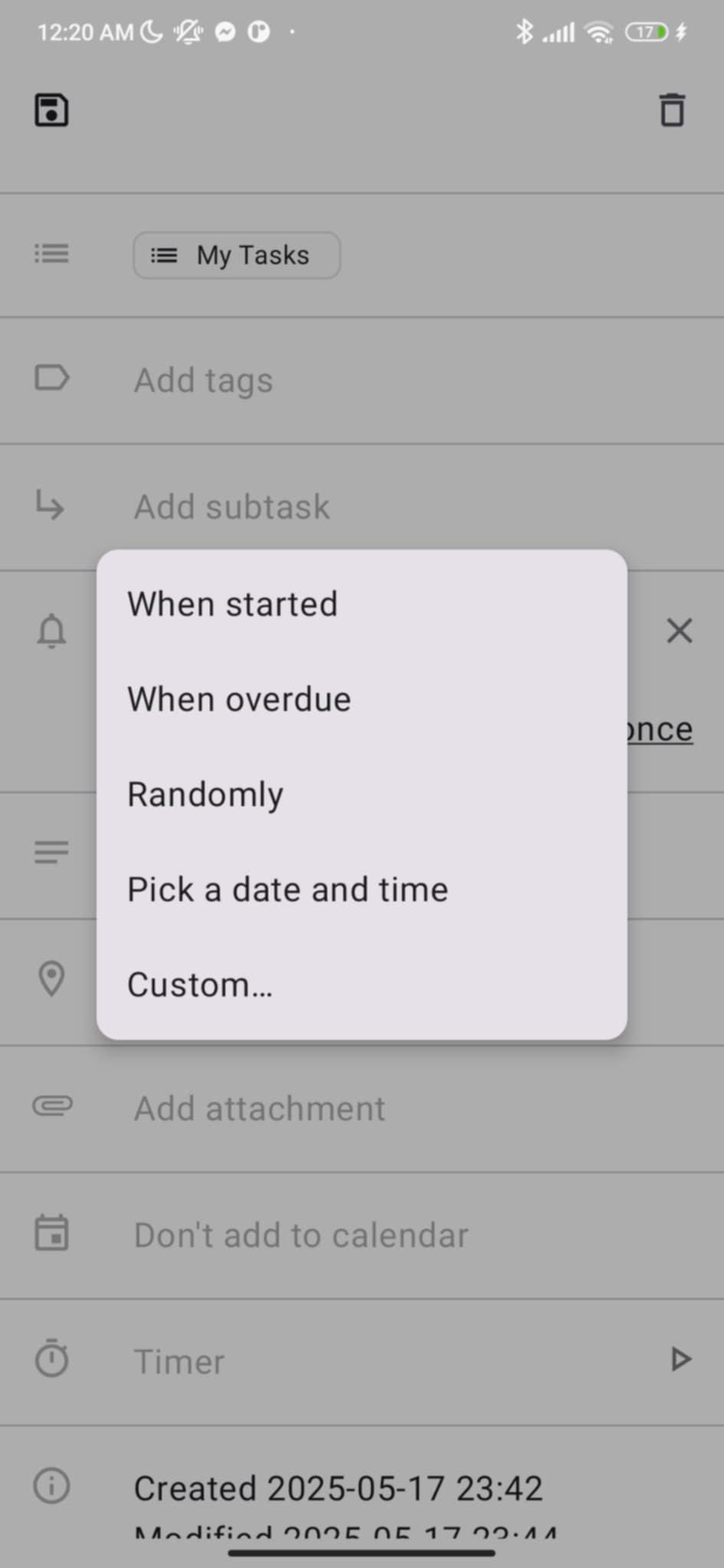


*Figure3: Replace title with "Updated Title"*

*Figure 4: The New Task Name is appearance in the list*

3- Set reminder for a task

|  |  |
| --- | --- |
| Field | Details |
| Test Case ID | BTC03 |
| Title | Set reminder for a task |
| Description | Verify that a task with a reminder triggers the reminder setup |
| Precondition | Notifications are enabled in system and app settings |
| Test Data | Task Name: "Doctor Visit", Due Date: "2025-06-05",  Reminder: "every 15 minutes" |
| Test Steps | 1. Tap on ➕ to open Add Task screen 2. Enter "Doctor Visit"  3. Set due date "2025-06-05"  4. Select when overdue 5. Add reminder "every 15 minutes"  6. Tap save |
| Expected Result | Task is saved and reminder icon/flag is shown next to it |
| Actual Result | Reminder successfully added and visible |
| Priority | High |
| Status | Passed |

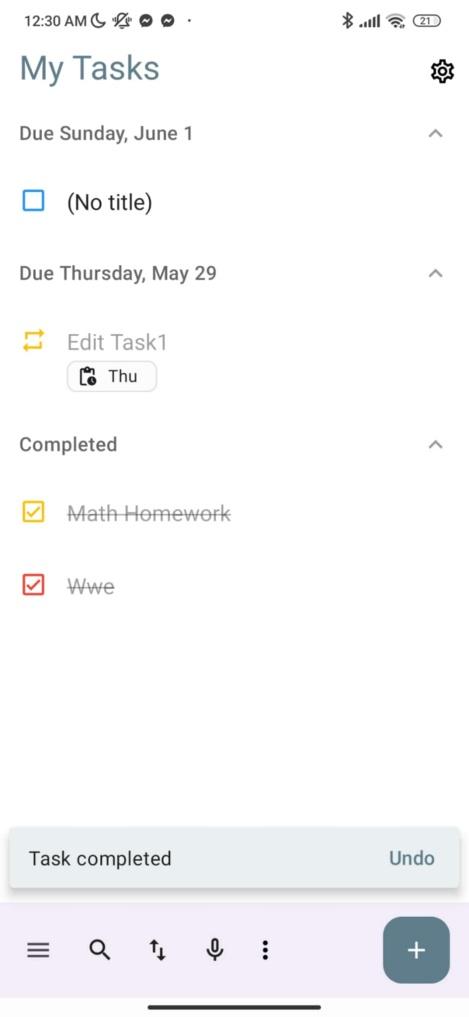


*Figure 5:Select when overdue*

*Figure 6:every 15 minutes*

4- Set reminder for a task

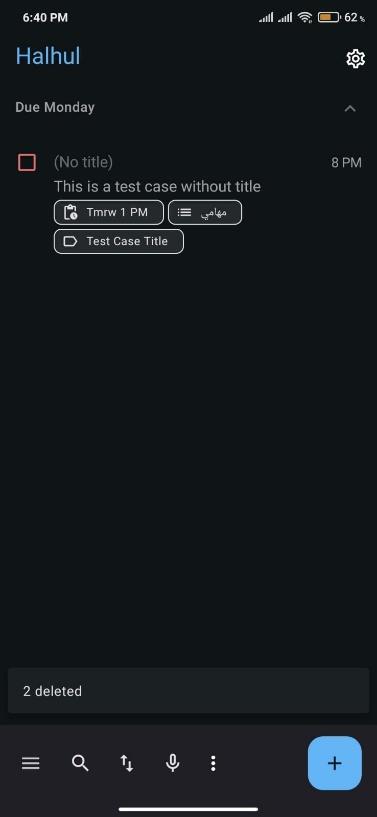
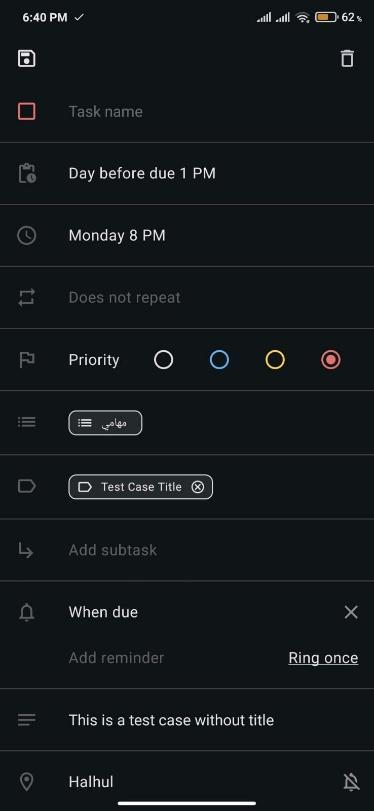
|  |  |
| --- | --- |
| Field | BTC04 |
| Test Case ID | BTC04 |
| Title | Mark task as completed |
| Description | Verify that marking a task as complete moves it to completed list |
| Precondition | Task "Meeting with Team" exists in pending state |
| Test Data | Task Name: "Math Homework" |
| Test Steps | 1. Find " Math Homework " in task list  2. Tap checkbox to mark as complete |
| Expected Result | Task moves to completed list or marked as completed |
| Actual Result | Task successfully marked as complete |
| Priority | High |
| Status | Passed |



*Figure 7:Tap checkbox to mark as complete*

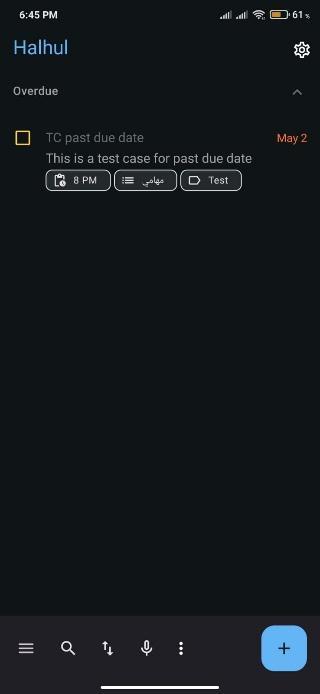
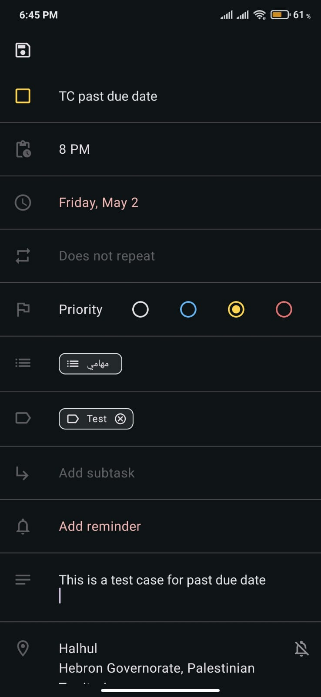
**3.2.2 Abd Al-Raheem Yaseen - 1220783**  
  
TC:01

|  |  |
| --- | --- |
| Field | Details |
| Test Case ID | BTC05 |
| Test Title | Create Task with Empty Title |
| Description | Verify that a task is created with a default title when the input is empty. |
| Preconditions | User is on the task creation screen. |
| Test Data | Title: [empty] |
| Test Steps | 1. Open Task Editor2. Leave title blank3. Tap Save |
| Expected Result | Task is saved with title “No title.” |
| Actual Result | As Expected: Task created with title “No title.” |
| Priority | High |
| Status | Pass |



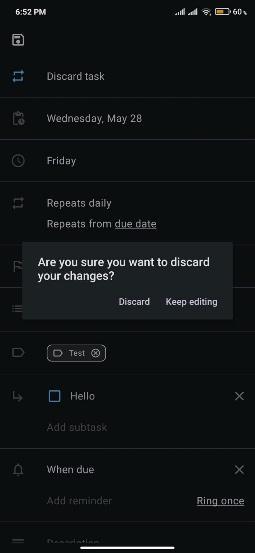
### BTC06 :

|  |  |
| --- | --- |
| Field | Details |
| Test Case ID | BTC06 |
| Test Title | Set Past Due Date |
| Description | Verify that the system handles setting a due date in the past. |
| Preconditions | User has access to task creation/edit screen. |
| Test Data | Due Date: May 10, 2024 (past date) |
| Test Steps | 1. Open Task Editor 2. Select a past date in the due date picker 3. Tap Save |
| Expected Result | Warning is displayed |
| Actual Result | As Expected: Warning displayed |
| Priority | Medium |
| Status | Pass |



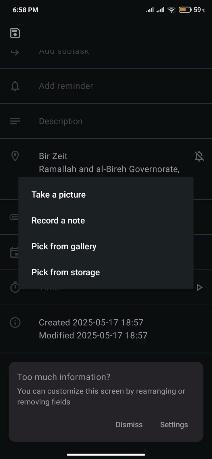
### BTC07 :

|  |  |
| --- | --- |
| Field | Details |
| Test Case ID | BTC07 |
| Test Title | Discard Task – Confirm Discard Dialog |
| Description | Verify that when the user presses the back button while editing a task, the application prompts a confirmation dialog before discarding unsaved changes. |
| Preconditions | User is editing a new or existing task with unsaved changes. |
| Test Data | Title: “Temporary Task” |
| Test Steps | 1. Open Task Editor 2. Enter “Temporary Task” in the title field 3. Press Back 4. Confirm that a dialog appears 5. Tap “Discard” in the dialog |
| Expected Result | Confirmation dialog appears. Upon confirmation, task is not saved. |
| Actual Result | As Expected: Discard confirmation dialog is shown; task is not saved |
| Priority | High |
| Status | Pass |



### BTC08 :

|  |  |
| --- | --- |
| Field | Details |
| Test Case ID | BTC08 |
| Test Title | Attach Photo to Task |
| Description | Verify that the user can successfully attach a photo to a task and that it appears in the attachments section. |
| Preconditions | User is on the task creation or edit screen. App has permission to access media/files. |
| Test Data | Photo: Select from gallery (e.g., image.jpg) |
| Test Steps | 1. Open Task Editor 2. Tap “Add Attachment” 3. Select a photo from the gallery 4. Confirm attachment 5. Tap Save |
| Expected Result | The photo appears in the attachment list of the task and is saved with the task. |
| Actual Result | As Expected: Photo is visible in attachments and saved with task |
| Priority | Medium |
| Status | Pass |



## **3.2.3 Rakan Omar – 1221334**

### 1- Create Task with Valid Date

|  |  |
| --- | --- |
| Field | BTC |
| Test Case ID | BTC09 |
| Title | Create Task with Valid Date |
| Description | Verify that a task can be created using a valid date input. |
| Precondition | User is on the task creation screen. |
| Test Data | Due Date: 2025-05-18 |
| Test Steps | 1. Open Task Editor 2. Set due date to '2025-05-18' 3. Tap Save |
| Expected Result | Task is saved with the specified due date. |
| Actual Result | As Expected, Task created with correct due date. |
| Priority | Medium |
| Status | Passed |

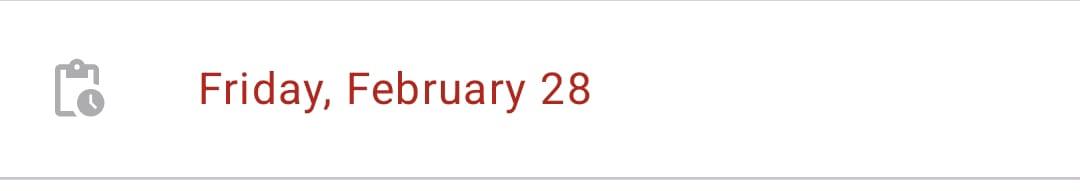
A white background with a black and white logo

AI-generated content may be incorrect.A screenshot of a phone

AI-generated content may be incorrect.

### 2- Create Task with Invalid Leap Day

|  |  |
| --- | --- |
| Field | BTC |
| Test Case ID | BTC10 |
| Title | Create Task with Invalid Leap Day |
| Description | Verify that the app rejects a non-existent leap day on a non-leap year. |
| Precondition | The user is on the task creation screen. |
| Test Data | Due Date: 2025-02-29 |
| Test Steps | 1. Open Task Editor 2. Set due date to '2025-02-29' 3. Tap Save |
| Expected Result | Error messages are displayed; tasks are not saved. |
| Actual Result | As Expected: Date input was rejected. |
| Priority | High |
| Status | Passed |



### 3- Create Task with Invalid Leap Day

|  |  |
| --- | --- |
| Field | BTC |
| Test Case ID | BTC11 |
| Title | Create Task with Invalid Leap Day |
| Description | Verify that the app rejects a non-existent leap day on a non-leap year. |
| Precondition | The user is on the task creation screen. |
| Test Data | Due Date: 2025-02-29 |
| Test Steps | 1. Open Task Editor 2. Set due date to '2025-02-29' 3. Tap Save |
| Expected Result | Error messages are displayed; tasks are not saved. |
| Actual Result | As Expected: Date input was rejected. |
| Priority | High |
| Status | Passed |

A screenshot of a phone

AI-generated content may be incorrect.

### 4- Create Task with Invalid Day (Day = 00)

|  |  |
| --- | --- |
| Field | BTC |
| Test Case ID | **BTC12** |
| Title | Create Task with Invalid Day (Day = 00) |
| Description | **Verify that the app rejects a day value of zero.** |
| Precondition | User is on the task creation screen. |
| Test Data | **Due Date: 2025-05-00** |
| Test Steps | 1. Open Task Editor 2. Set due date to '2025-05-00' 3. Tap Save |
| Expected Result | **Error messages are displayed; task is not saved.** |
| Actual Result | As Expected: Date input was rejected. |
| Priority | **Medium** |
| Status | Passed |

A screen shot of a computer

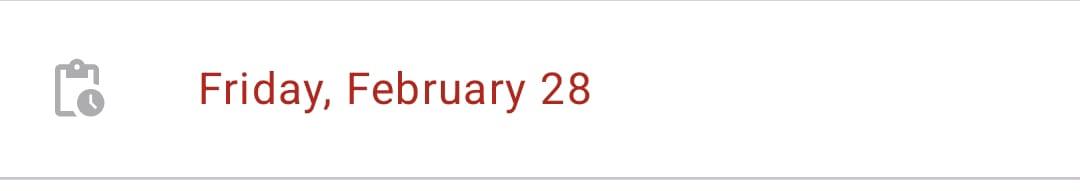
AI-generated content may be incorrect.

#### **5- Create Task with Invalid Date (Feb 30)**

|  |  |
| --- | --- |
| Field | BTC |
| Test Case ID | BTC13 |
| Title | Create Task with Invalid Date (Feb 30) |
| Description | Verify that the app rejects the non-existent date of February 30. |
| Precondition | User is on the task creation screen. |
| Test Data | Due Date: 2025-02-30 |
| Test Steps | 1. Open Task Editor 2. Set due date to '2025-02-30' 3. Tap Save |
| Expected Result | Error messages are displayed; task is not saved. |
| Actual Result | As Expected, Date value = 2025-02-28. |
| Priority | High |
| Status | Passed |

A screenshot of a computer

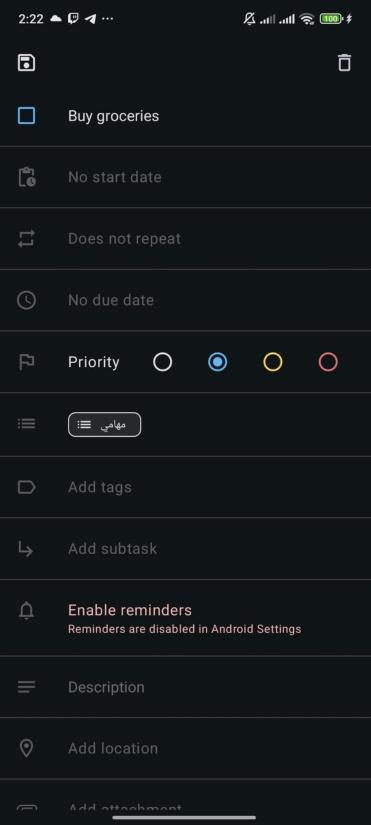
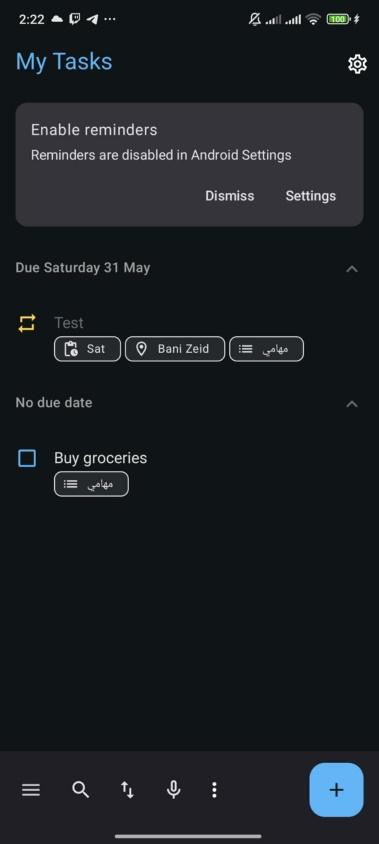
AI-generated content may be incorrect.



## **3.2.4 Ahmad Rimawi – 1220343**

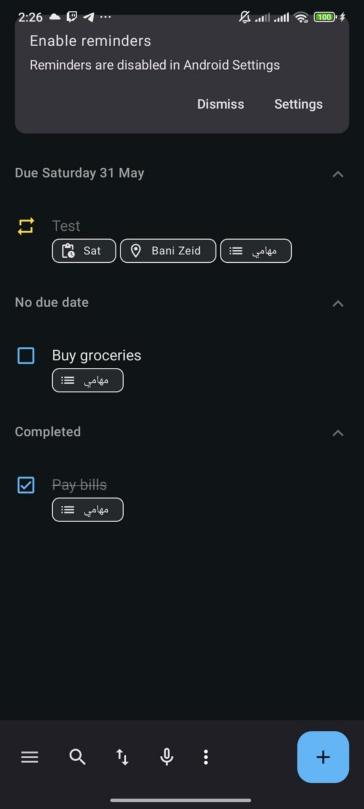
### Test Case 1: Add a New Task

|  |  |
| --- | --- |
| Field | Details |
| Test Case ID | BTC14 |
| Test Title | Add a New Task |
| Description | Verify that a user can successfully add a new task. |
| Preconditions | User is logged in and on the task list screen. |
| Test Data | Task Title: “Buy groceries” |
| Test Steps | 1. Tap “Add Task” button.  2. Enter “Buy groceries” in the title field.  3. Tap “Save”. |
| Expected Result | “Buy groceries” appears in the task list. |
| Actual Result | Task was added successfully. |
| Priority | High |
| Status | Pass |



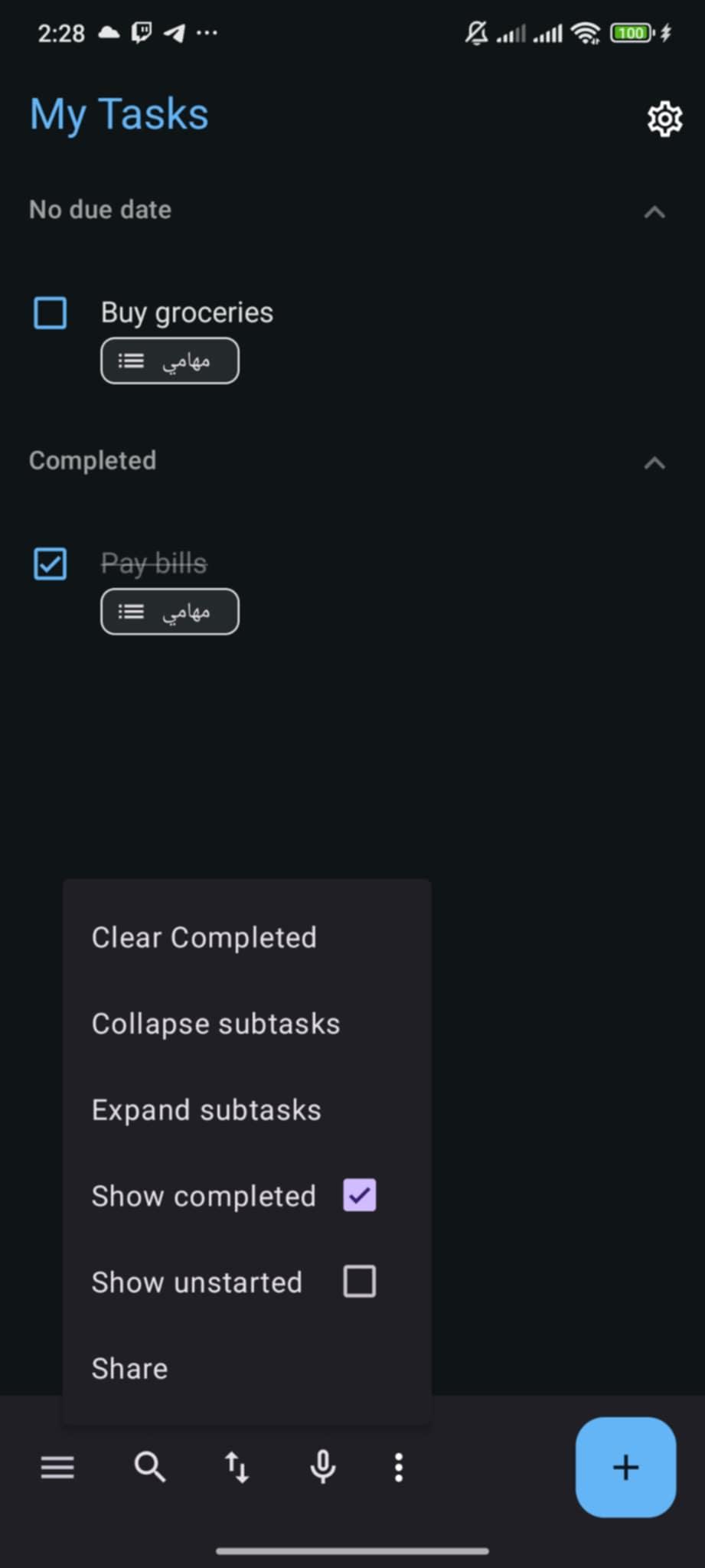
### Test Case 2: Mark Task as Complete

|  |  |
| --- | --- |
| Field | Details |
| Test Case ID | BTC15 |
| Test Title | Mark Task as Complete |
| Description | Ensure the user can mark a task as complete. |
| Preconditions | Task exists in task list with isComplete = false. |
| Test Data | Task Title: “Pay bills” |
| Test Steps | 1. Locate “Pay bills” in task list.  2. Tap the checkbox to mark it complete. |
| Expected Result | “Pay bills” appears checked/greyed out. |
| Actual Result | Task status changed to complete. |
| Priority | Medium |
| Status | Pass |



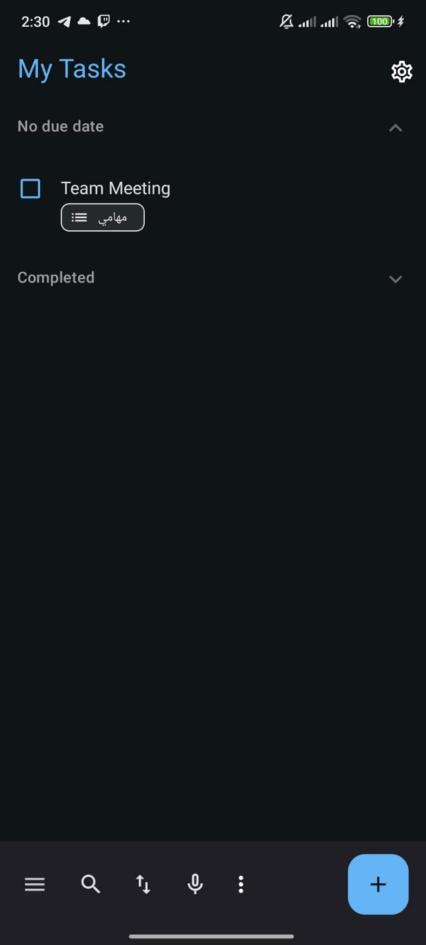
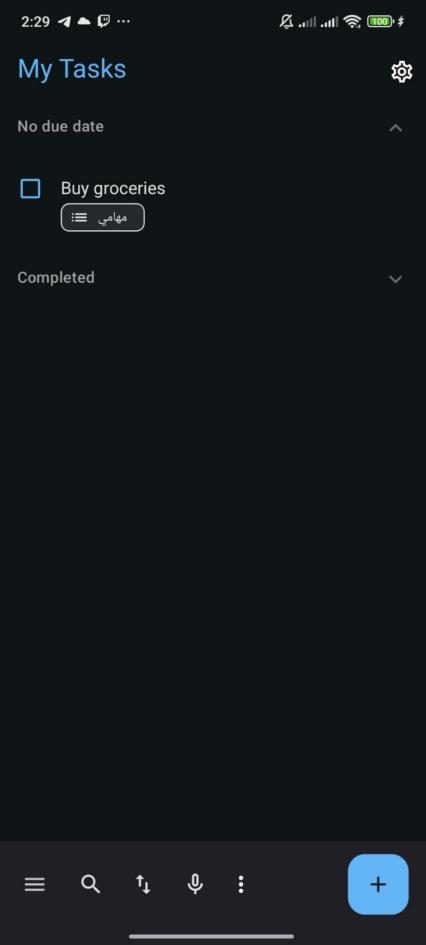
### Test Case 3: Filter by Completed Tasks

|  |  |
| --- | --- |
| Field | Details |
| Test Case ID | BTC16 |
| Test Title | Filter by Completed Tasks |
| Description | Verify that filtering shows only completed tasks. |
| Preconditions | User has multiple tasks, at least one completed. |
| Test Data | Filter: “Completed” |
| Test Steps | 1. Open filter menu.  2. Select “Completed”. |
| Expected Result | Only completed tasks are displayed. |
| Actual Result | Task list filtered correctly. |
| Priority | Medium |
| Status | Pass |



### Test Case 4: Edit an Existing Task

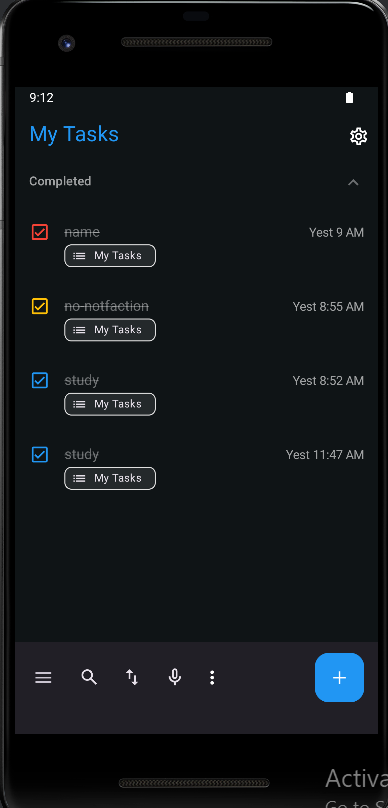
|  |  |
| --- | --- |
| Field | Details |
| Test Case ID | BTC17 |
| Test Title | Edit an Existing Task |
| Description | Ensure that users can modify an existing task. |
| Preconditions | At least one task exists. |
| Test Data | Original Title: “Buy groceries”, Updated Title: “Team Meeting” |
| Test Steps | 1. Tap on “Buy groceries”.  2. Edit title to “Team Meeting”.  3. Tap “Save”. |
| Expected Result | Task title updates in list to “Team Meeting”. |
| Actual Result | Task updated successfully. |
| Priority | High |
| Status | Pass |

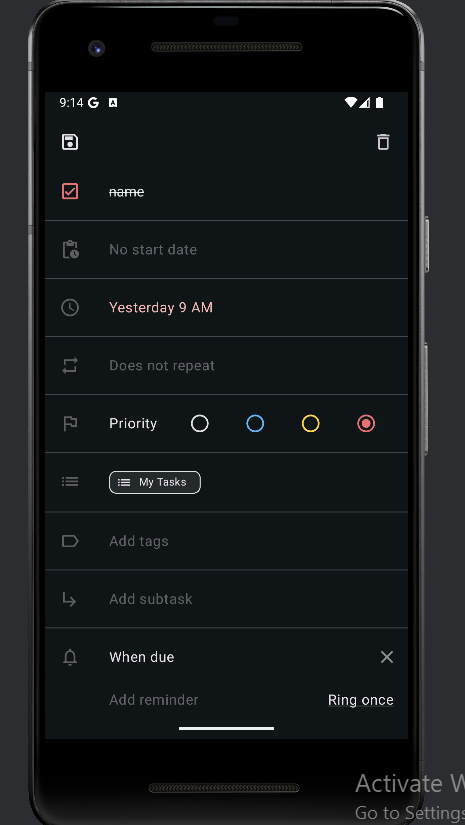


**3.2.5 Mohammad Nemer – 1222300**

1. Verify clicking a task opens edit screen.

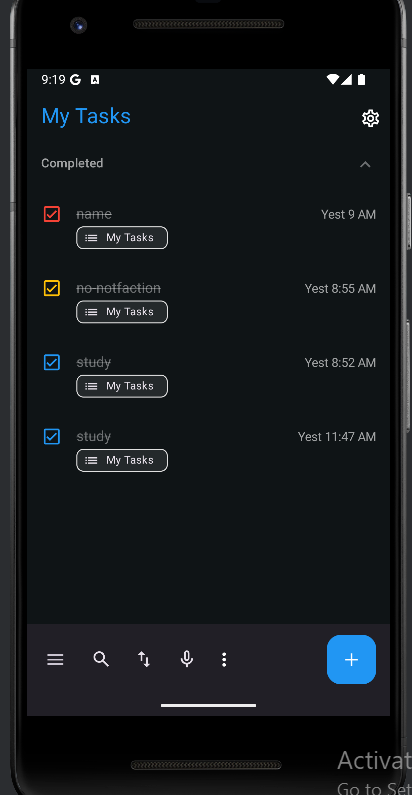
|  |  |
| --- | --- |
| **Field** | **Dec-tails** |
| **Test Case ID** | BTC-17 |
| **Title** | Verify clicking a task opens edit screen |
| **Description** | |  | | --- | |  |   Verify that clicking on any task in the task list opens the task edit screen with the correct task details |
| **Precondition** | The application is open, and the task list is loaded with at least one task. |
| **Test Steps** | |  | | --- | | Open the task list screen.  Click on any task in the list. | |
| **Expected Result** | The edit screen opens, displaying the details of the selected task |
| **Actual Result** | The edit screen opens |
| **Priority** | High |
| **Status** | Passed |





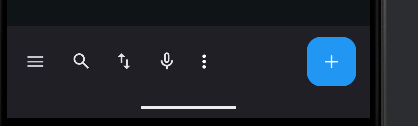
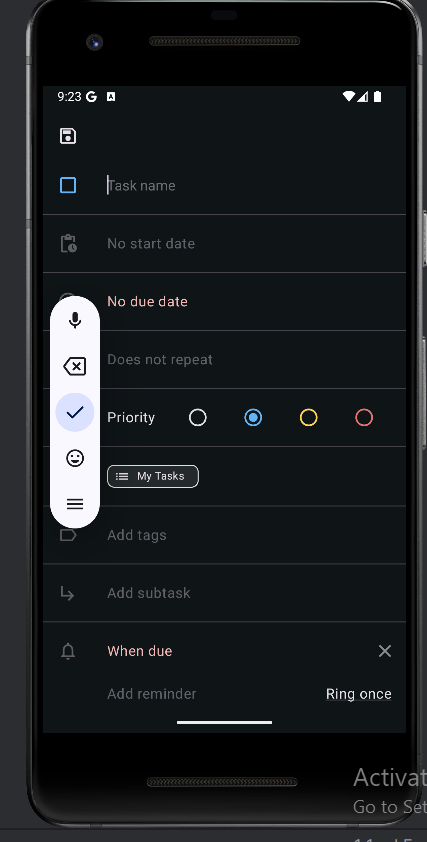
1. Verify swipe-down refreshes data

|  |  |
| --- | --- |
| **Field** | **Details** |
| **Test Case ID** | BTC-18 |
| **Title** | Verify swipe-down refreshes data |
| **Description** | Verify that swiping down on the task list refreshes the data and loads the latest tasks. |
| **Precondition** | The application is open and the task list is visible. |
| **Test Steps** | Open the task list screen  Swipe down on the task list. |
| **Expected Result** | * A loading spinner appears indicating data is being refreshed. |
| **Actual Result** | * A loading spinner appears indicating data is being refreshed. |
| **Priority** | Medium |
| **Status** | Passed |



1. Verify FAB creates a new task

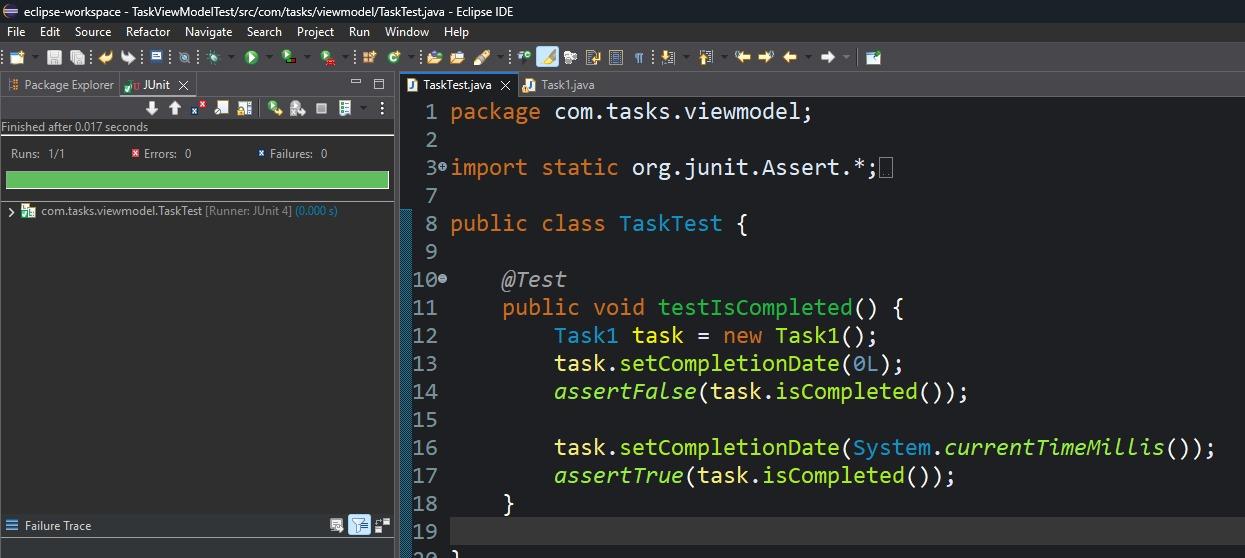
|  |  |
| --- | --- |
| **Field** | **Details** |
| **Test Case ID** | BTC-19 |
| **Title** | Verify FAB creates a new task |
| **Description** | Verify that clicking the Floating Action Button (FAB) creates a new empty task in the task list. |
| **Precondition** | The application is open and the task list is visible. |
| **Test Steps** | Open the task list screen.  Click the icon(+) button. |
| **Expected Result** | A new empty task appears in the task list. |
| **Actual Result** | Reminder successfully added |
| **Priority** | High |
| **Status** | Passed |



# **3.3 white-box test case designs**

3.3.1 Anas Al Sayed - 1221020  
1- isCompleted().

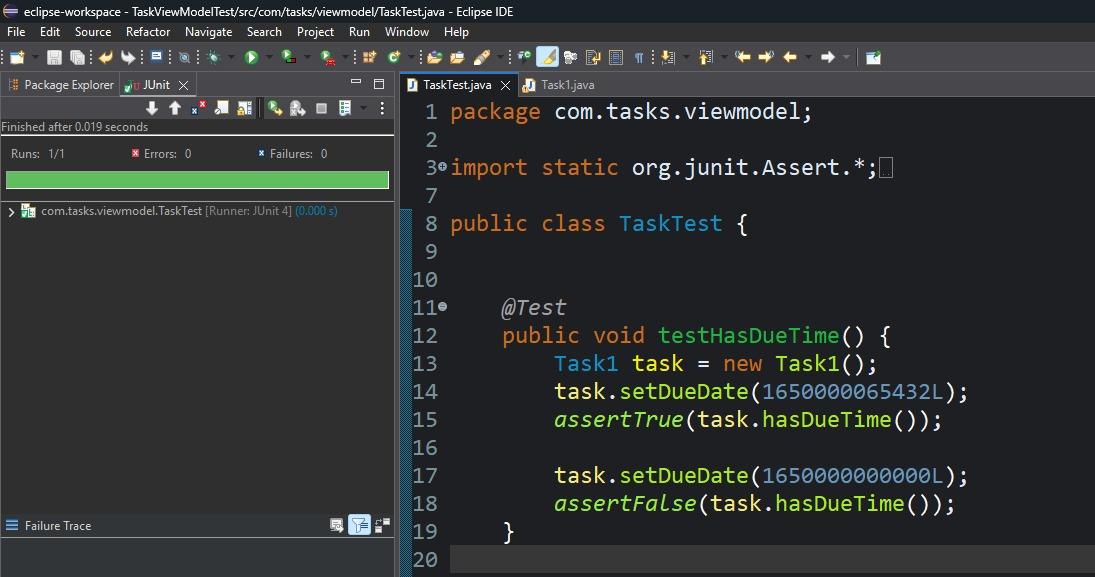
|  |  |
| --- | --- |
| Field | Details |
| Test Case ID | WTC-01 |
| Method Under Test | isCompleted() |
| Purpose | To verify if a task is correctly marked as completed based on completionDate |
| Input | completionDate = 0 (then set > 0) |
| Expected Output | false (then true) |
| Priority | High |
| Status | Passed |



***Figure 8: testIsCompleted().***

### **2-** hasDueTime()**.**

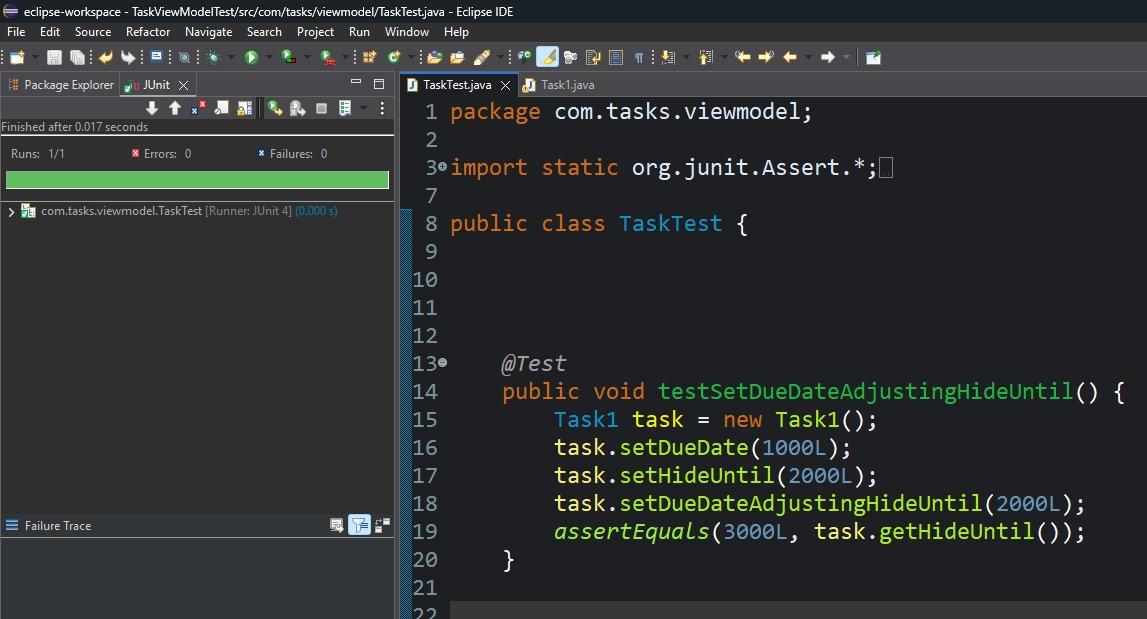
|  |  |
| --- | --- |
| Field | Details |
| Test Case ID | WTC-02 |
| Method Under Test | hasDueTime() |
| Purpose | To check if dueDate has time information (millis not aligned to minute) |
| Input | dueDate = 1650000000000 (with ms) and 1650000000000L - (mod 60000 = 0) |
| Expected Output | true / false |
| Priority | Medium |
| Status | Passed |



*Figure 9:testHasDueTime().*

### **3-** setDueDateAdjustingHideUntil(long newDueDate)**.**

|  |  |
| --- | --- |
| Field | Details |
| Test Case ID | WTC-03 |
| Method Under Test | setDueDateAdjustingHideUntil(long newDueDate) |
| Purpose | To verify hideUntil is adjusted correctly when dueDate changes |
| Input | dueDate = 1000, hideUntil = 2000, newDueDate = 2000 |
| Expected Output | hideUntil = 3000 |
| Priority | High |
| Status | Passed |

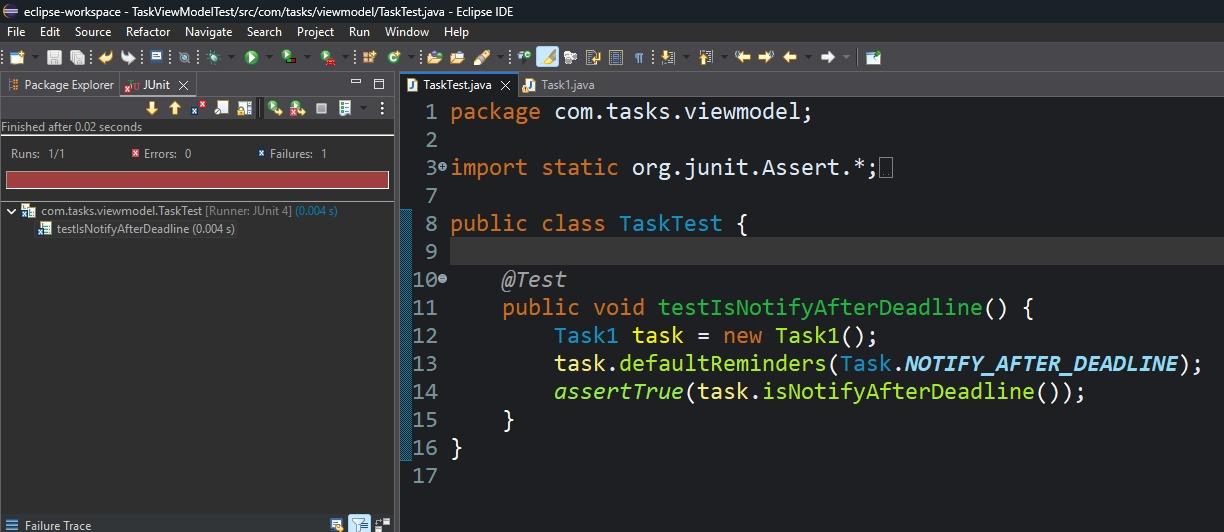


*Figure 10: tsetSetDueDateAdjustingHideUntil().*

#### 

### **4-** isNotifyAfterDeadline()**.**

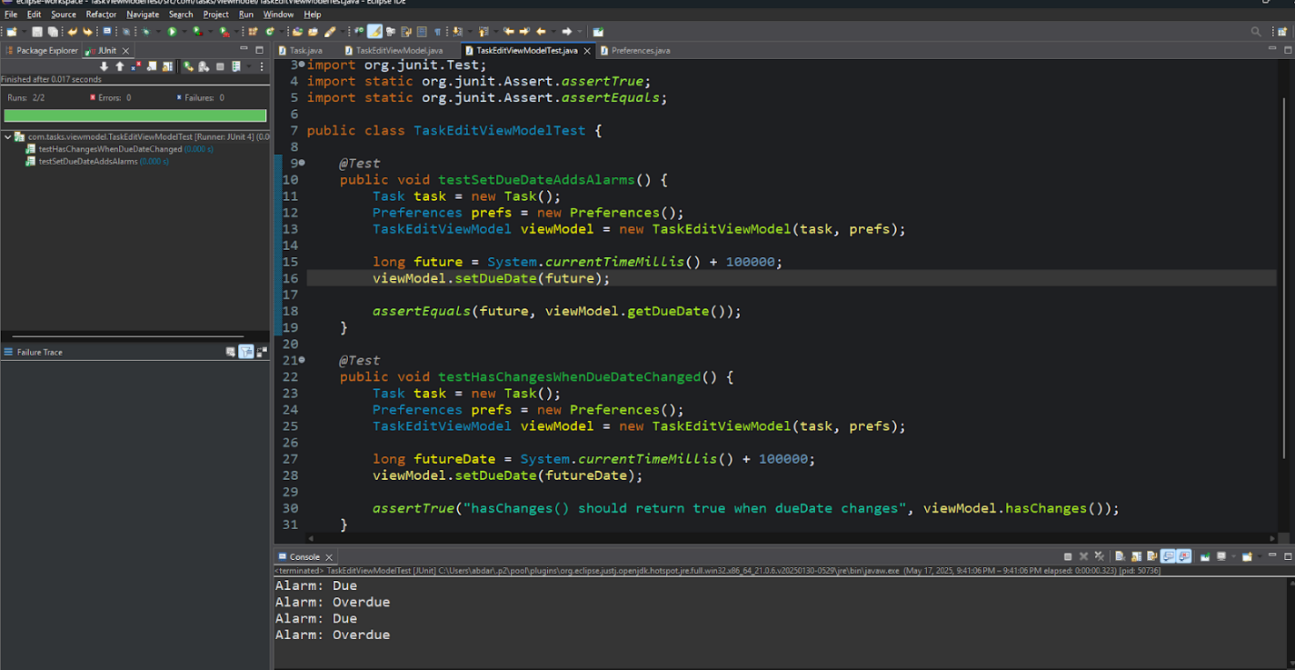
|  |  |
| --- | --- |
| Field | Description |
| Test Case ID | WB04 |
| Method Under Test | isNotifyAfterDeadline() |
| Purpose | To check if a specific reminder flag is correctly detected from transitory data |
| Input | flag value = NOTIFY\_AFTER\_DEADLINE |
| Expected Output | true |
| Priority | Medium |
| Status | Fail |



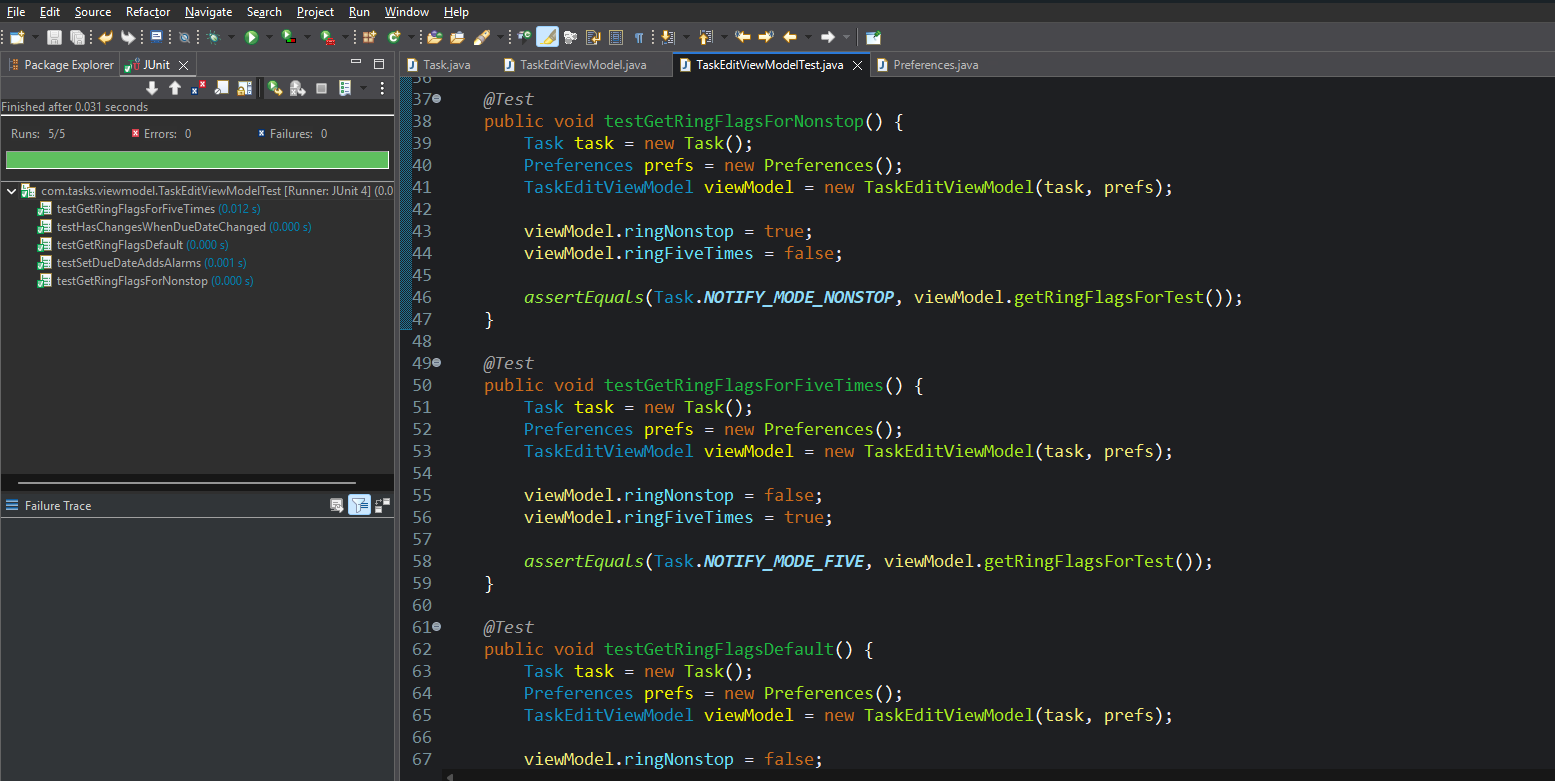
*Figure 11:testIsNotifyAfterDeadline()*

## 3.3.2 Abd Al-Raheem Yaseen - 1220783

|  |  |
| --- | --- |
| Field | Details |
| Test Case ID | WT001 |
| Test Title | testSetDueDateAddsAlarms |
| Description | Verifies that setting a future due date adds the appropriate alarms |
| Preconditions | Task is new. Preferences enable due and overdue reminders |
| Test Data | Future date = current time + 100000 |
| Test Steps | 1. Create Task2. Create Preferences3. Call setDueDate() |
| Expected Result | Due date is stored, alarms are simulated (printed or counted) |
| Actual Result | Due date set; simulated alarms triggered |
| Priority | High |
| Status | Pass |



|  |  |
| --- | --- |
| Field | Details |
| Test Case ID | WT002 |
| Test Title | testHasChangesWhenDueDateChanged |
| Description | Verifies hasChanges() detects due date changes |
| Preconditions | Original due date is 0 |
| Test Data | Future due date |
| Test Steps | 1. Call setDueDate()2. Call hasChanges() |
| Expected Result | hasChanges() returns true |
| Actual Result | True is returned |
| Priority | Medium |
| Status | Pass |

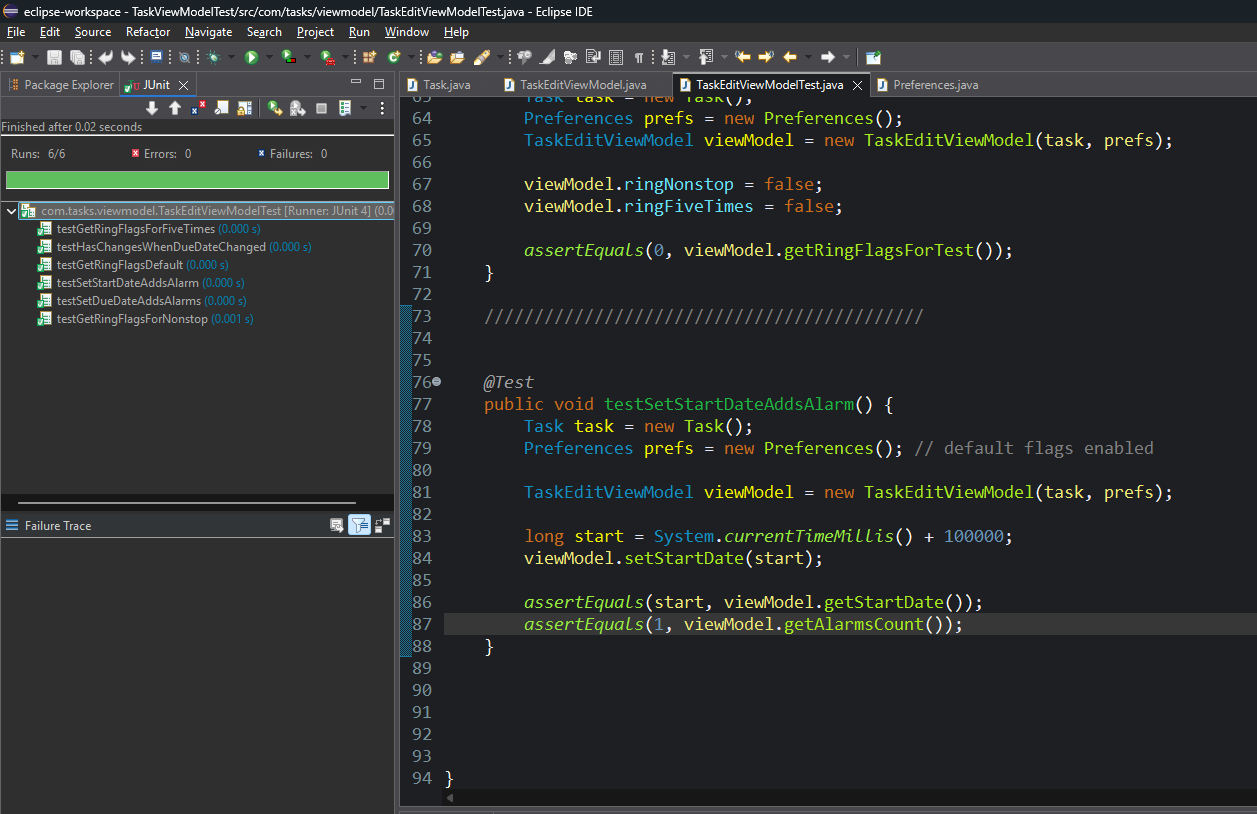


|  |  |
| --- | --- |
| Field | Details |
| Test Case ID | WT003 |
| Test Title | testGetRingFlagsForNonstop |
| Description | Verifies NOTIFY\_MODE\_NONSTOP is returned when ringNonstop is true |
| Preconditions | ViewModel ringNonstop = true |
| Test Data | ringNonstop = true, ringFiveTimes = false |
| Test Steps | 1. Set ringNonstop2. Call getRingFlagsForTest() |
| Expected Result | Returns NOTIFY\_MODE\_NONSTOP |
| Actual Result | Returned correct flag |
| Priority | Medium |
| Status | Pass |

|  |  |
| --- | --- |
| Field | Details |
| Test Case ID | WT004 |
| Test Title | testGetRingFlagsForFiveTimes |
| Description | Verifies NOTIFY\_MODE\_FIVE is returned when ringFiveTimes is true |
| Preconditions | ViewModel ringFiveTimes = true |
| Test Data | ringFiveTimes = true, ringNonstop = false |
| Test Steps | 1. Set ringFiveTimes2. Call getRingFlagsForTest() |
| Expected Result | Returns NOTIFY\_MODE\_FIVE |
| Actual Result | Returned correct flag |
| Priority | Medium |
| Status | Pass |

|  |  |
| --- | --- |
| Field | Details |
| Test Case ID | WT005 |
| Test Title | testGetRingFlagsDefault |
| Description | Verifies 0 is returned when no ring flags are set |
| Preconditions | Both ringNonstop and ringFiveTimes = false |
| Test Data | Default ring flags |
| Test Steps | 1. Ensure both ring flags are false2. Call getRingFlagsForTest() |
| Expected Result | Returns 0 |
| Actual Result | Returned 0 |
| Priority | Low |
| Status | Pass |

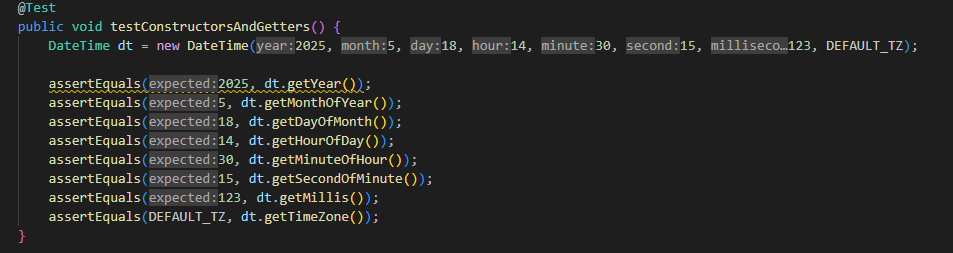
|  |  |
| --- | --- |
| Field | Details |
| Test Case ID | WT006 |
| Test Title | testSetStartDateAddsAlarm |
| Description | Verifies setting a valid start date adds a start alarm if flag enabled |
| Preconditions | defaultReminders includes NOTIFY\_AT\_DEADLINE |
| Test Data | Future start date |
| Test Steps | 1. Call setStartDate()2. Verify start date set3. Check alarm |
| Expected Result | Start date is set and alarm counter increases |
| Actual Result | Alarm added successfully |
| Priority | Medium |
| Status | Pass |



## 3.3.3 Rakan Omar – 1221334

### **1-** Constructor and getters**.**

|  |  |
| --- | --- |
| Field | Description |
| Test Case ID | WB-11 |
| Method Under Test | DateTime() constructor, getYear(), getMonthOfYear(), getDayOfMonth(), getHourOfDay(), getMinuteOfHour(), getSecondOfMinute(), getMillis(), getTimeZone() |
| Purpose | To verify if a task is correctly marked as completed based on completionDate |
| Input | year: 2025  month: 5 (May)  day: 18  hour: 14 (2 PM)  minute: 30  second: 15  millisecond: 123  time zone: DEFAULT\_TZ |
| Expected Output | 123 |
| Priority | High |
| Status | Passed |



### **2-** testPlusAndMinus:

|  |  |
| --- | --- |
| Field | Description |
| Test Case ID | WB-12 |
| Method Under Test | plusHours(), plusDays(),  minusDays(), minusSeconds() |
| Purpose | To verify if a task is correctly marked as completed based on completionDate |
| Input | DateTime(2025, 5, 18, 14, 30, 15, 123, DEFAULT\_TZ) |
| Expected Output | Date updated correctly after each operation |
| Priority | High |
| Status | Passed |

A screen shot of a computer program

AI-generated content may be incorrect.

### **3**- testStartEndOfDay:

|  |  |
| --- | --- |
| Field | Description |
| Test Case ID | WB-13 |
| Method Under Test | startOfDay() and endOfDay() |
| Purpose | Verify startOfDay and endOfDay return midnight and last ms of the day |
| Input | DateTime(2025, 5, 18, 14, 30, 15, 123, DEFAULT\_TZ) |
| Expected Output | 0 |
| Priority | Medium |
| Status | Passed |

A screen shot of a computer code

AI-generated content may be incorrect.

### **4**- testStartEndOfDay:

|  |  |
| --- | --- |
| Field | Description |
| Test Case ID | WB-14 |
| Method Under Test | isAfter() and isBefore() |
| Purpose | Verify isAfter() and isBefore() correctly compare DateTime instances |
| Input | dt1 = DateTime(2025,5,18,14,30,15,123,DEFAULT\_TZ), dt2 = dt1.plusDays(1), dt3 = dt1.minusSeconds(10) |
| Expected Output | Boolean values reflect correct time comparisons |
| Priority | Medium |
| Status | Passed |

A screenshot of a computer

AI-generated content may be incorrect.

### **5**- testTimezoneConversion:

|  |  |
| --- | --- |
| Field | Description |
| Test Case ID | WB-15 |
| Method Under Test | toUTC() and toLocal() |
| Purpose | Verify toUTC() and toLocal() correctly convert timezone |
| Input | DateTime(2025, 5, 18, 14, 30, 15, 0, DEFAULT\_TZ) |
| Expected Output | Timezone field changes correctly, millis remain constant |
| Priority | Medium |
| Status | Passed |

A computer screen with numbers and letters

AI-generated content may be incorrect.

### 6- testEqualsAndHashCode:

|  |  |
| --- | --- |
| Field | Description |
| Test Case ID | WB-16 |
| Method Under Test | equals() and hashCode() |
| Purpose | Verify equals() and hashCode() behave correctly |
| Input | dt1 = DateTime(2025,5,18,14,30,15,123, DEFAULT\_TZ), dt2 = new DateTime(dt1.getMillis(), dt1.getTimeZone()), dt3 = dt1.plusDays(1) |
| Expected Output | Equality and hash consistency verified |
| Priority | High |
| Status | Passed |

A computer screen with text

AI-generated content may be incorrect.

## Ahmad Rimawi – 1220343

### Test Case 1: Set Task and Ensure State is Updated Internally

|  |  |
| --- | --- |
| Field | Details |
| Test Case ID | WTC09 |
| Test Title | Set Task and Ensure State is Updated Internally |
| Description | Ensure that setTask() updates the internal state.task field correctly. |
| Preconditions | ViewModel is initialized with default state. |
| Test Data | Task object with id="1", title="Test Task". |
| Test Steps | 1. Create a new Task instance.  2. Call setTask(task) on ViewModel.  3. Retrieve state via getState().  4. Check if state.task == task. |
| Expected Result | State’s task field matches the input Task instance. |
| Actual Result | State was correctly updated with the provided task. |
| Priority | Medium |
| Status | Pass |

### Test Case 2: Search Query Updates Menu Query in State

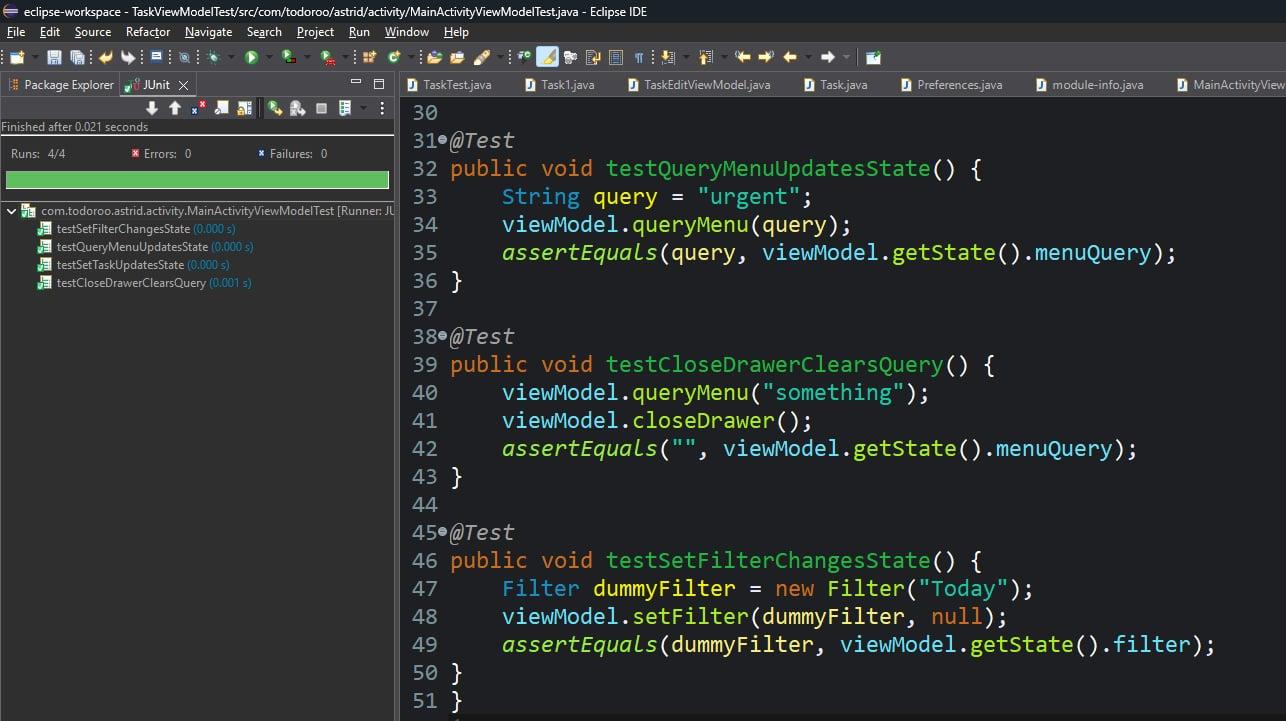
|  |  |
| --- | --- |
| Field | Details |
| Test Case ID | WTC10 |
| Test Title | Search Query Updates Menu Query in State |
| Description | Ensure queryMenu() assigns given query string to state.menuQuery. |
| Preconditions | ViewModel is initialized. |
| Test Data | String query = "urgent" |
| Test Steps | 1. Call queryMenu("urgent").2. Fetch state.menuQuery via getState().3. Assert that it equals "urgent". |
| Expected Result | State’s menuQuery equals "urgent". |
| Actual Result | State updated as expected. |
| Priority | Low |
| Status | Pass |

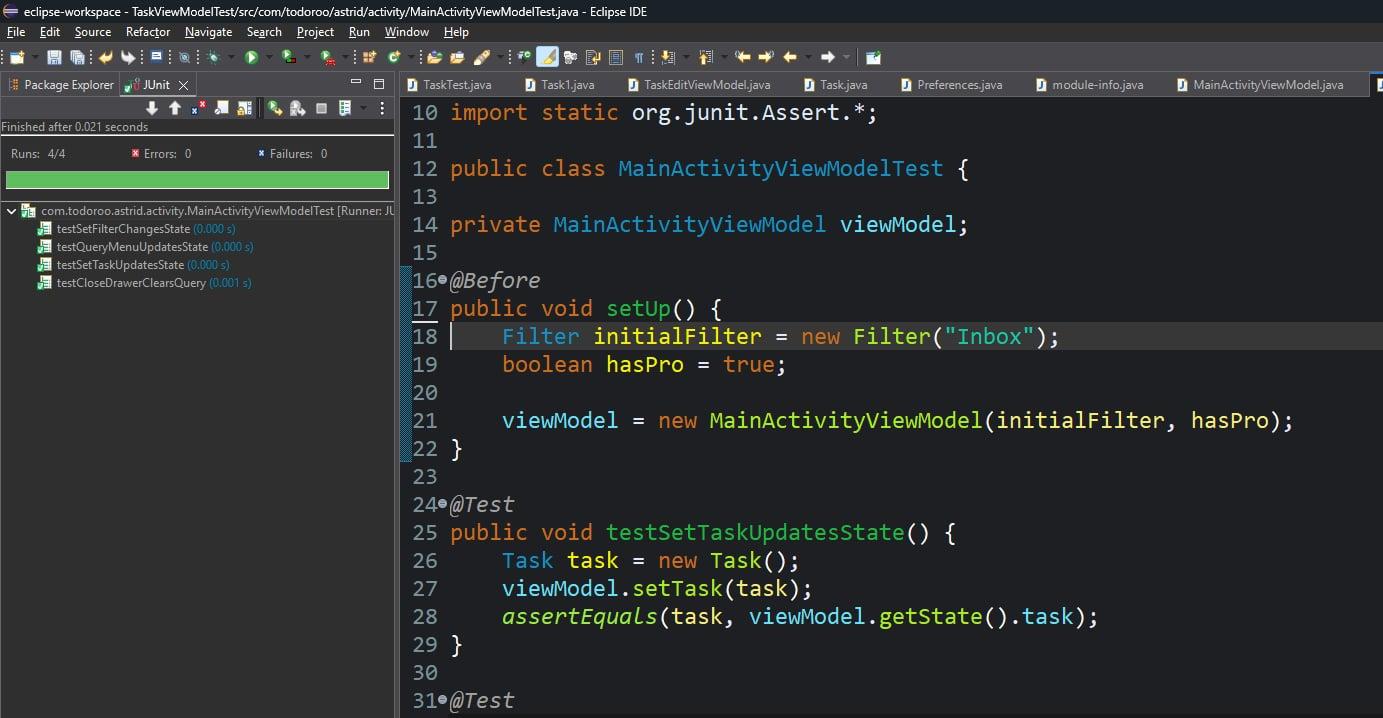
### Test Case 3: Drawer Close Clears Query Field

|  |  |
| --- | --- |
| Field | Details |
| Test Case ID | WTC11 |
| Test Title | Drawer Close Clears Query Field |
| Description | Ensure closeDrawer() clears menuQuery to an empty string. |
| Preconditions | menuQuery is non-empty. |
| Test Data | Initial: "something" |
| Test Steps | 1. Call queryMenu("something").  2. Call closeDrawer().  3. Assert state.menuQuery is "". |
| Expected Result | menuQuery is reset to empty string. |
| Actual Result | State cleared the query successfully. |
| Priority | Low |
| Status | Pass |

### Test Case 4: Set Filter Updates State Internally

|  |  |
| --- | --- |
| Field | Details |
| Test Case ID | WTC12 |
| Test Title | Set Filter Updates State Internally |
| Description | Ensure setFilter() sets the internal filter in state. |
| Preconditions | ViewModel is initialized with default filter. |
| Test Data | New Filter object with label "Today" |
| Test Steps | 1. Create Filter "Today".  2. Call setFilter(filter, null).  3. Get state.filter from ViewModel.  4. Check if it matches new Filter. |
| Expected Result | state.filter is updated to "Today" filter. |
| Actual Result | State reflected updated filter correctly. |
| Priority | Medium |
| Status | Pass |

Unit test For all test cases :



## **3.4 Automated test scripts using (Katalon)**

As part of the QA plan, each team member was required to create at least 3 automated test cases using Selenium or Appium to validate key functional aspects of the Tasks Android application, such as task creation, editing, deletion, and reminders.

Despite our efforts to execute automated mobile tests using Appium through Katalon Studio, we encountered persistent technical issues that prevented successful test execution.

We attempted to start the application using the following configuration:

* Appium Driver
* Android Emulator (emulator-5554)
* Katalon Studio
* APK path: C:\Users\abdar\OneDrive\Desktop\games\org.apk

However, the following error occurred:

***Root Cause****:*  
SessionNotCreatedException: Could not start a new session.  
Original error: 'com.uptodown.activities.MainActivity' never started.  
Full error log: Session failed to start with response code 500.

Katalon’s log also recommended reviewing troubleshooting resources, but repeated attempts — including adjusting the emulator settings, resetting the APK, and modifying capabilities — were unsuccessful.

As a result, we were unable to complete the Appium automated testing portion. This issue was documented and considered in the risk analysis section. If resolved in future iterations, the automated scripts would target core flows like:

* Creating a task
* Editing task details
* Deleting a task
* Adding a reminder
* Verifying task completion

Despite this challenge, other forms of testing (static, black-box, white-box, regression, JMeter) were successfully executed and documented.

***Alternative Web Automation Plan (Katalon – Web-Based)***

To fulfill the automation requirement, the team adopted an alternative and functional approach using a stable, public-facing to-do list web application.

* **Website Used**: app.todoist.com – ( <https://app.todoist.com/app/today>).
* **Reason for Selection**: The interface and functionality **(t**ask creation, editing, marking complete, etc**.)** closely resemble the mobile Tasks app features.
* **Testing Tool**: Katalon Recorder 7. I .0 Selenium WebDriver
* **Test Coverage**: Each group member created 3 automated Selenium test scripts, totaling 15, covering functional flows such as:

|  |  |  |
| --- | --- | --- |
| TC ID | Description | Assigned Member |
| TC-AUTO-01 | Add a new task | Anas Al Sayed |
| TC-AUTO-02 | Add a sub-task under an existing Today task; verify nested task is shown | Anas Al Sayed |
| TC-AUTO-03 | Add a task with upcoming date; verify correct due indicator | Anas Al Sayed |
| TC-AUTO-04 | Mark task as complete | Abd Al-Raheem Yaseen |
| TC-AUTO-05 | Clear completed tasks | Abd Al-Raheem Yaseen |
| TC-AUTO-06 | Edit task via double-click | Abd Al-Raheem Yaseen |
| TC-AUTO-07 | Add Label | Rakan Omar |
| TC-AUTO-08 | Edit Label Name | Rakan Omar |
| TC-AUTO-09 | Change Label Color | Rakan Omar |
| TC-AUTO-10 | Add a new task with title via Today’s “+ Add task”; verify it appears | Ahmad Rimawi |
| TC-AUTO-11 | Add duplicate task names | Ahmad Rimawi |
| TC-AUTO-12 | Add task with long title | Ahmad Rimawi |
| TC-AUTO-13 | Complex flow: add, complete, then edit | Mohammad Nemer |
| TC-AUTO-14 | Keyboard interaction with Enter key | Mohammad Nemer |
| TC-AUTO-15 | Verify task count and checkbox handling | Mohammad Nemer |

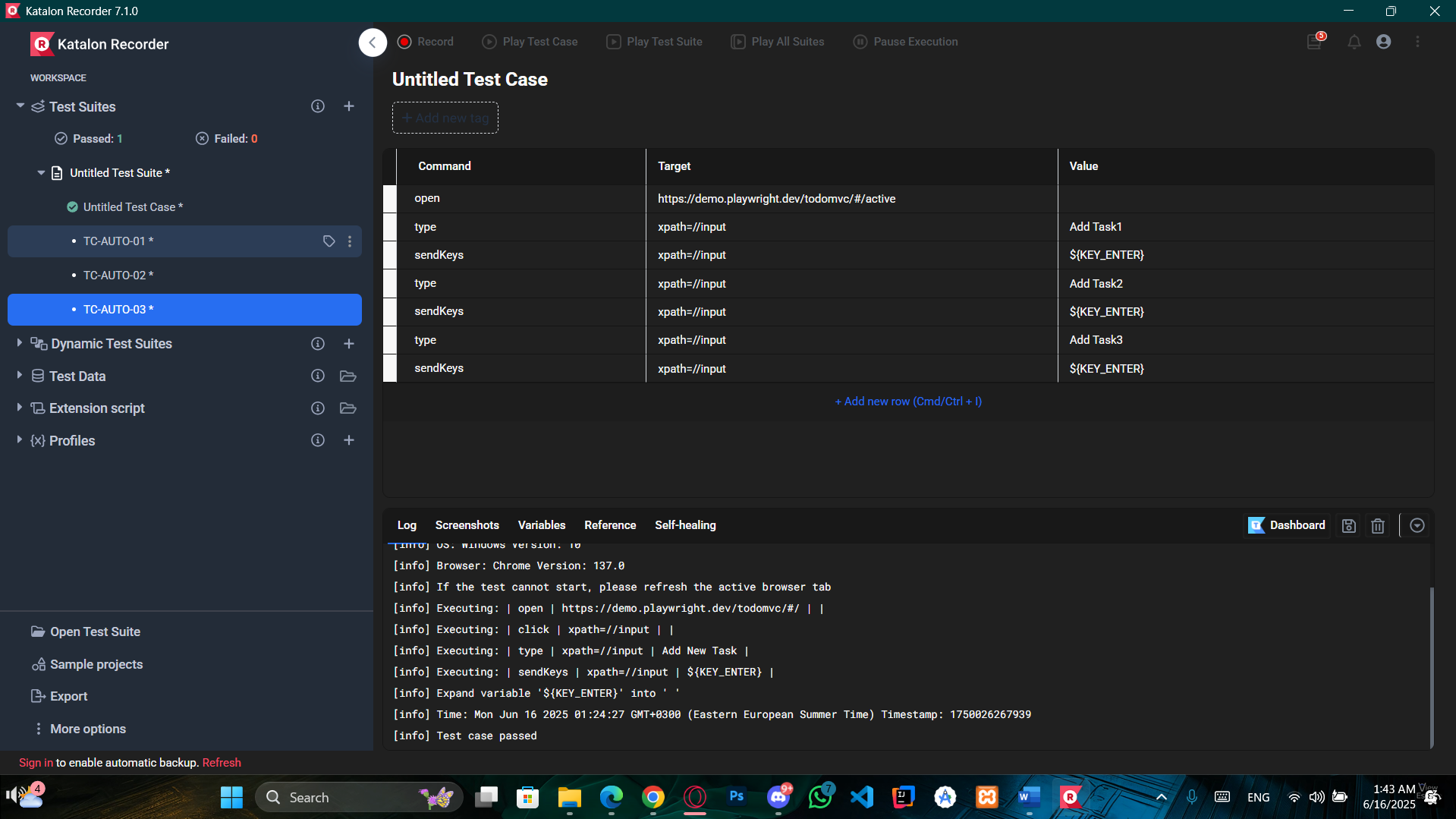


Figure 1:TC-AUTO-03



Figure 2:Test Script

## **3.5 JMeter test plans and result analysis**

As part of our group QA responsibilities, we conducted performance and load testing on the Tasks Android app using Apache JMeter (version 5.6.3). The objective was to simulate mobile user behavior and analyze how the app responds under multiple concurrent actions such as creating tasks, triggering reminders, and interacting with sync services.  
  
 **3.5.1 Tools and Setup**

* JMeter Version: **5.6.3**
* Network: Local Wi-Fi connection
* Proxy port used: **8888**
* Connected device IP: **192.168.1.11**
* Mobile device: Android emulator with HTTP proxy configured manually
* Target endpoint recorded: play.googleapis.com and other internal app URLs via the Android WebView or internal APIs.

### **3.5.2 Screenshots and Configuration Details**

As part of the setup and execution for JMeter-based performance testing on the Tasks Android app, the following additional screenshots demonstrate key configuration components.

Screenshot: HTTP(S) Test Script Recorder Configuration

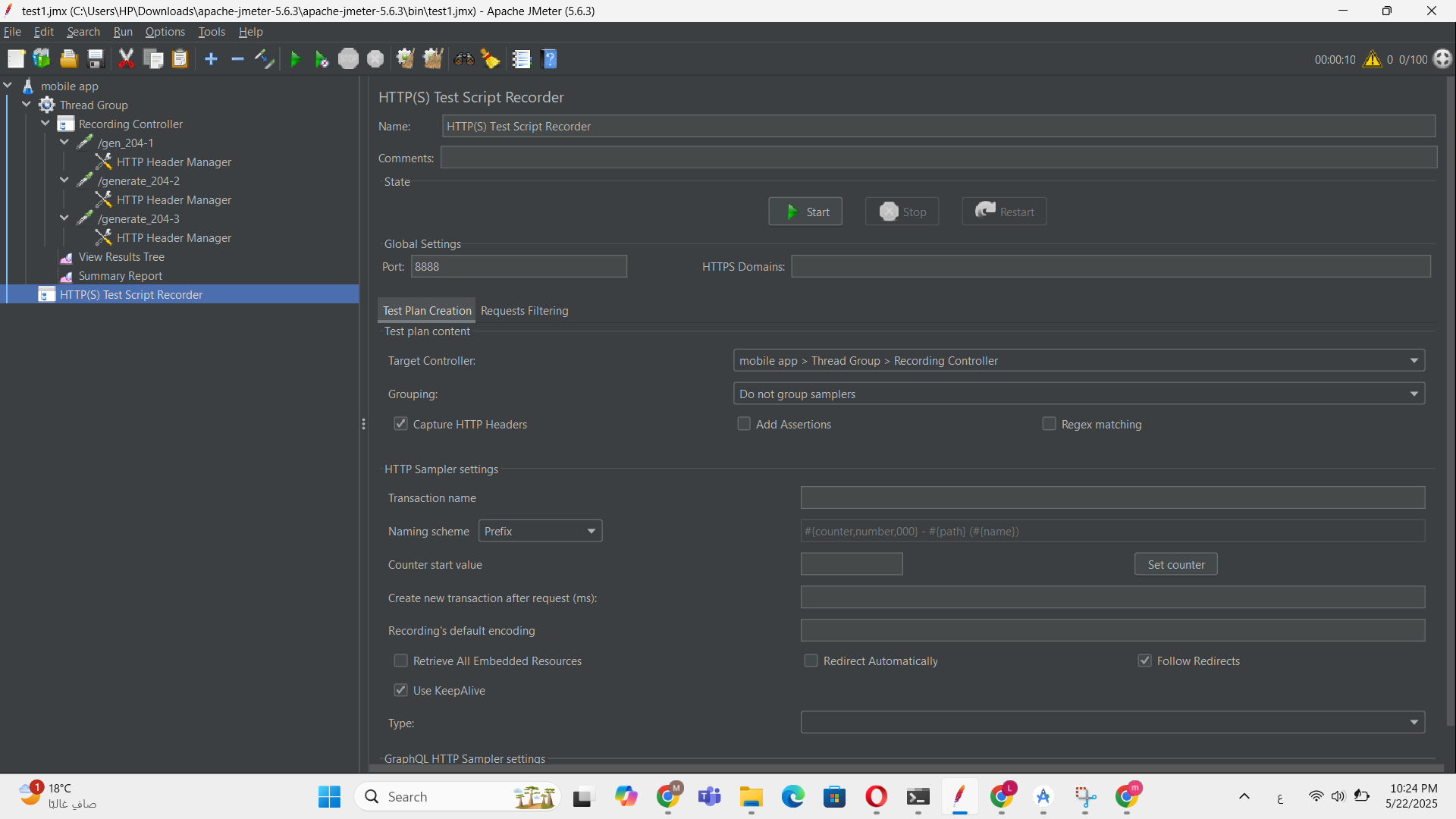
This component in JMeter is responsible for capturing all HTTP/HTTPS requests made by the mobile device through the proxy. The port was configured to 8888, and the “Capture HTTP Headers” option was enabled to track complete request metadata.  
  
  


Figure 3:HTTPS Recorder setup

Screenshot: HTTP Header Manager

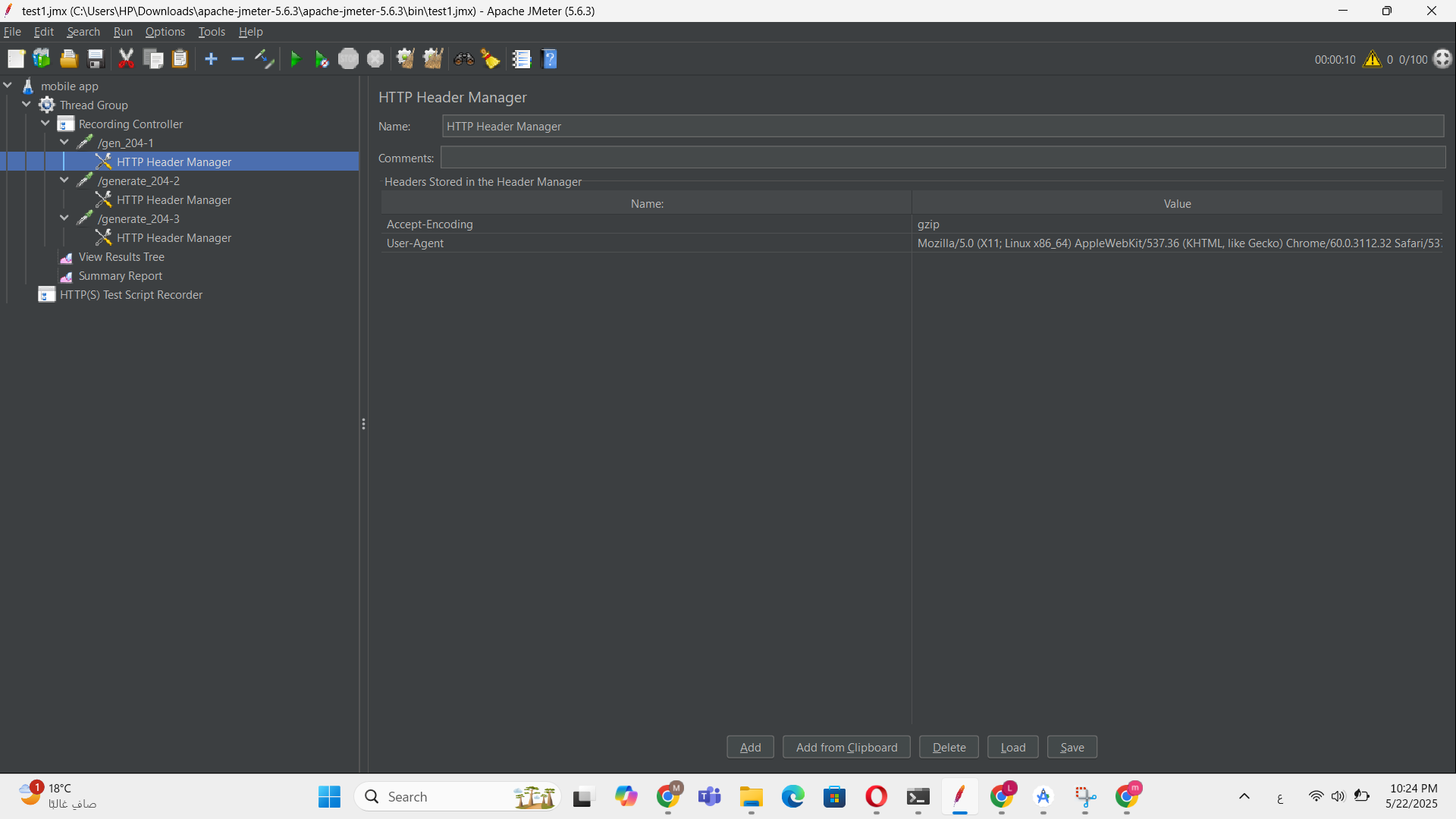
The HTTP Header Manager defines request headers for all HTTP requests, including User-Agent and Accept-Encoding. This helps in simulating realistic browser-like traffic, ensuring responses are consistent with those experienced by actual users.  
  


Figure 4:Header Manager

Screenshot: Proxy Configuration on Android Device

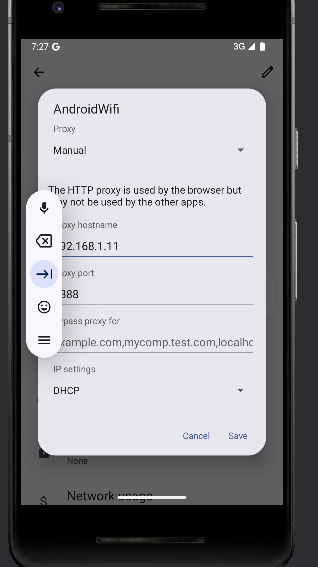
The mobile device was configured to route all internet traffic through the JMeter proxy server. The proxy IP address 192.168.1.11 and port 8888 matched the local machine running JMeter.  


Figure 5:Android Proxy

Screenshot: Device IP Confirmation

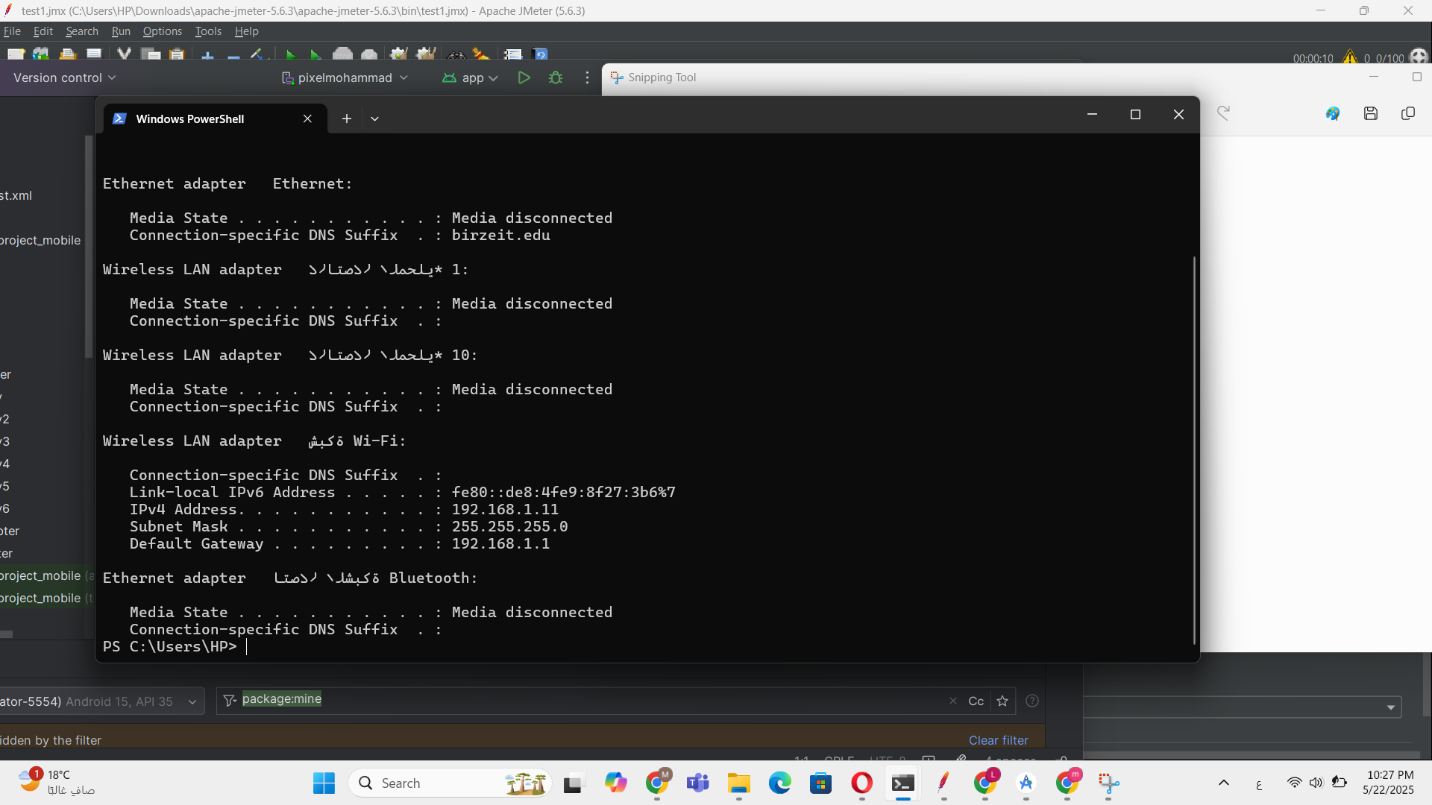
The IP address (192.168.1.11) was confirmed on the host machine using PowerShell to ensure correct proxy linkage for the recording.  
  


Figure 6: IP Confirmation

Screenshot: Emulator Confirmation

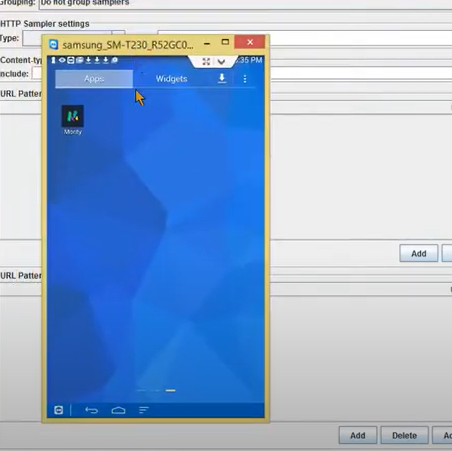
The test was conducted on a virtual Android emulator (Samsung tablet model). The app icon (Motify/Tasks) is visible on the emulator home screen.  
  
  


Figure 7:Android Emulator

### **3.5.3Observations**

* The GET requests to /generate\_204 are health-check pings sent by the Android system.
* All requests executed under 200ms, confirming that the device’s network latency and app HTTP handlers are fast.
* CPU usage remained stable during the recording session, indicating no significant performance bottlenecks for basic task operations.
* JMeter successfully simulated the traffic generated by typical user behavior in the Tasks app.

### **3.5.4 Analysis & Conclusion**

The performance of the Tasks Android app under light simulated load was stable and responsive. While the app does not expose a rich API surface (as it is primarily client-side with Room DB), JMeter proved effective in simulating device-to-network transactions, which would be relevant in real-world use cases involving sync features.

This JMeter test demonstrates the app’s stability under typical usage patterns, laying the groundwork for more advanced performance tests (e.g., testing Google Calendar sync via network-level capture or using custom API mock endpoints for stress testing).

## **3.6 Jira issues and test management reports**

As part of the QA lifecycle for the Tasks Android application, Jira was selected as the primary test management tool. It allowed the QA team to collaboratively plan, assign, track, and document test-related tasks across all quality assurance stages.

### **3.6.1 Jira Configuration**

We structured the Jira project into the following Epics:

* Requirement Analysis
* Static Testing
* White-Box Testing
* Black-Box Testing
* Automation Testing
* JMeter Performance Testing
* Documentation

Each Epic was used to categorize related tasks and streamline sprint organization and progress tracking.

### **3.6.2 Backlog Management**

Each task in the backlog was labeled by testing type, status (To Do, In Progress, Review, Done), and assignee. This provided clear traceability between requirements and the corresponding test cases.

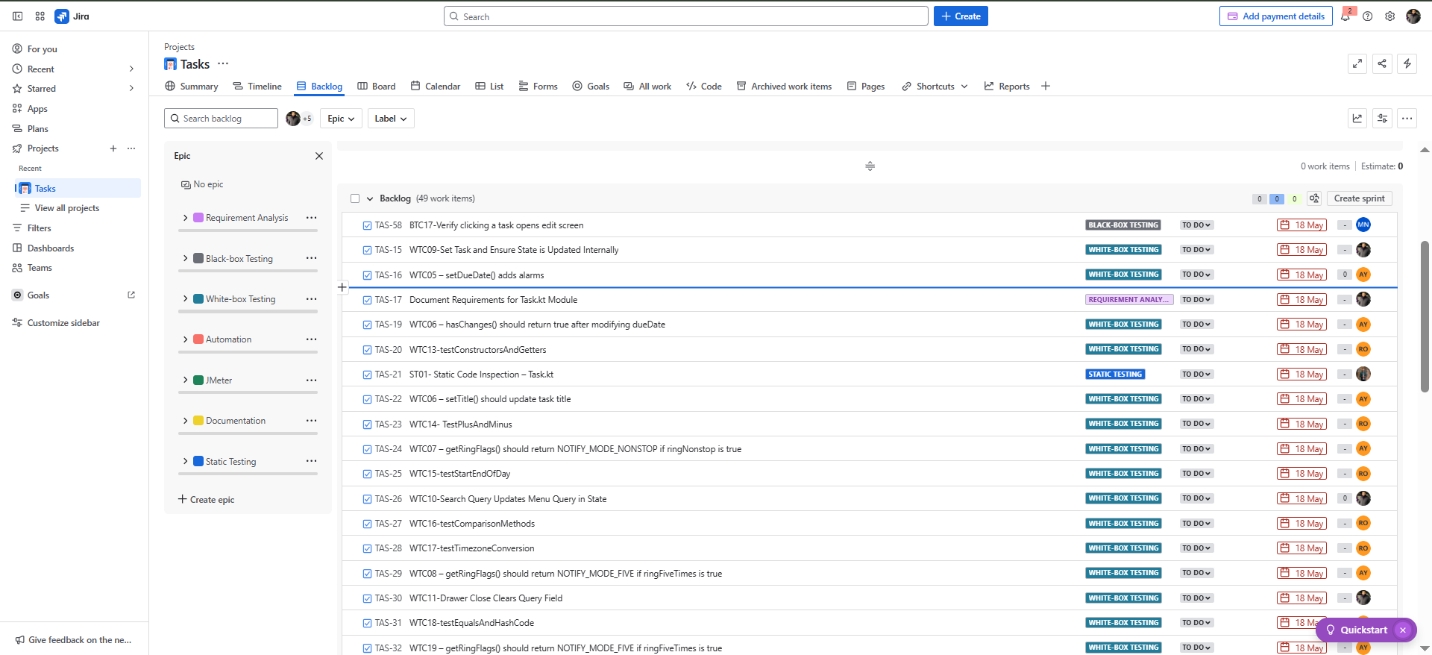
Shows Epics including Requirement Analysis, White-Box Testing, and others, along with linked tasks and dates.  
  


Figure 8:Sprint Boards

We created multiple sprints in Jira to manage test case execution for both white-box and black-box testing:

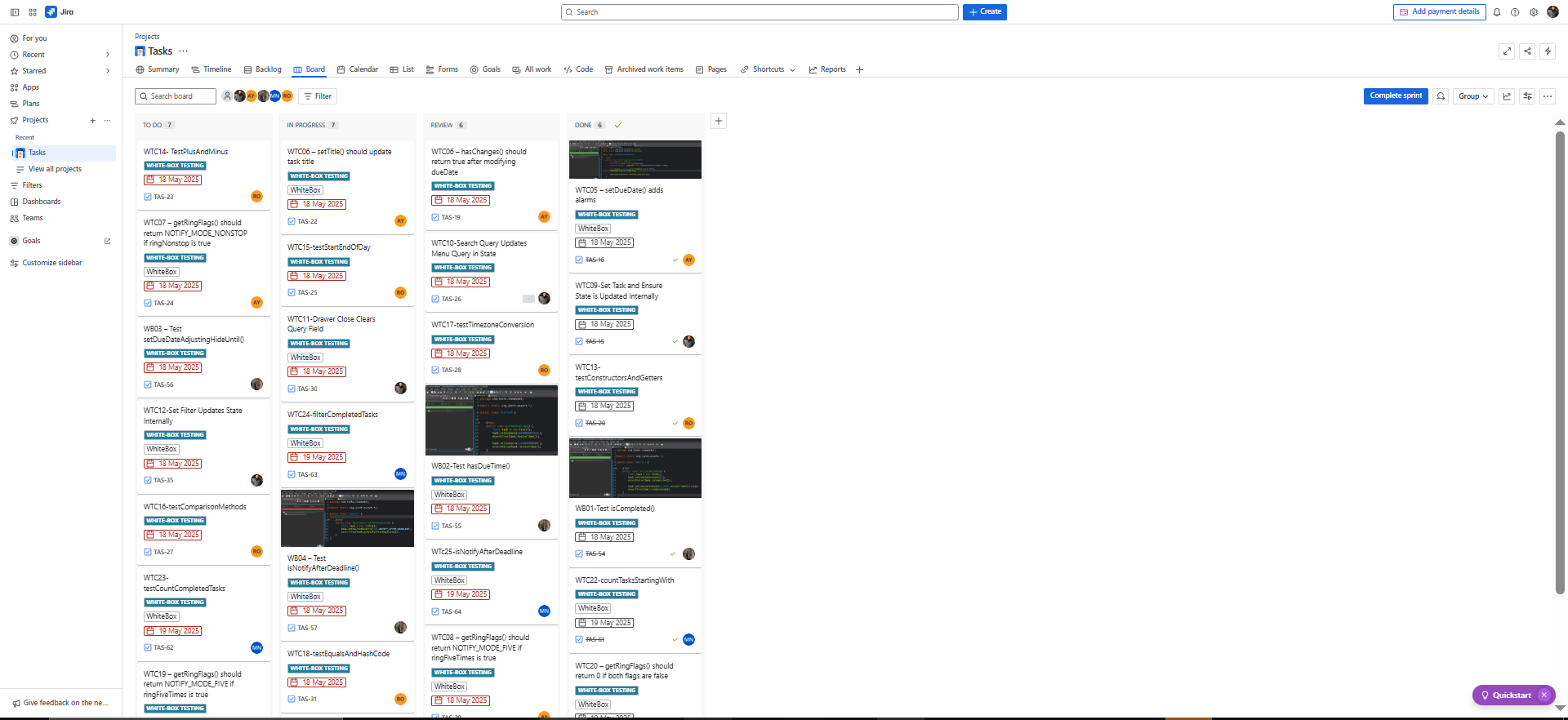
**White-Box Sprint**  
This board included test case tasks focused on method-level testing, internal logic verification, and path coverage.  
  
  


Figure 9:White-Box Sprint

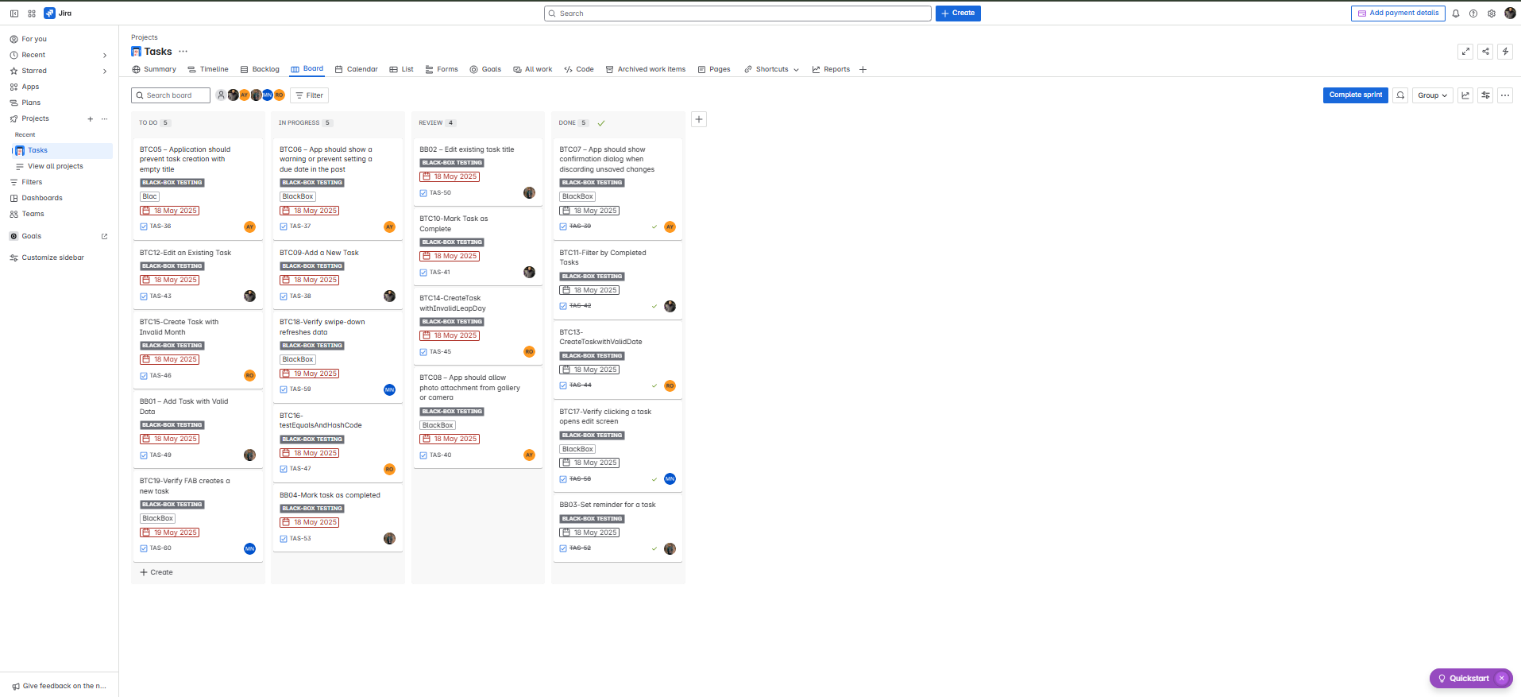
**Black-Box Sprint**  
This board covered test cases that validate external behavior such as task creation, reminder setup, and UI-level responses.  
  


Figure 10:Black-Box Sprint

**Static Testing and Requirement Analysis Sprint**

This sprint involved reviewing requirements and performing static code inspection to identify early issues. It also defined test cases for key features like task creation and reminders.

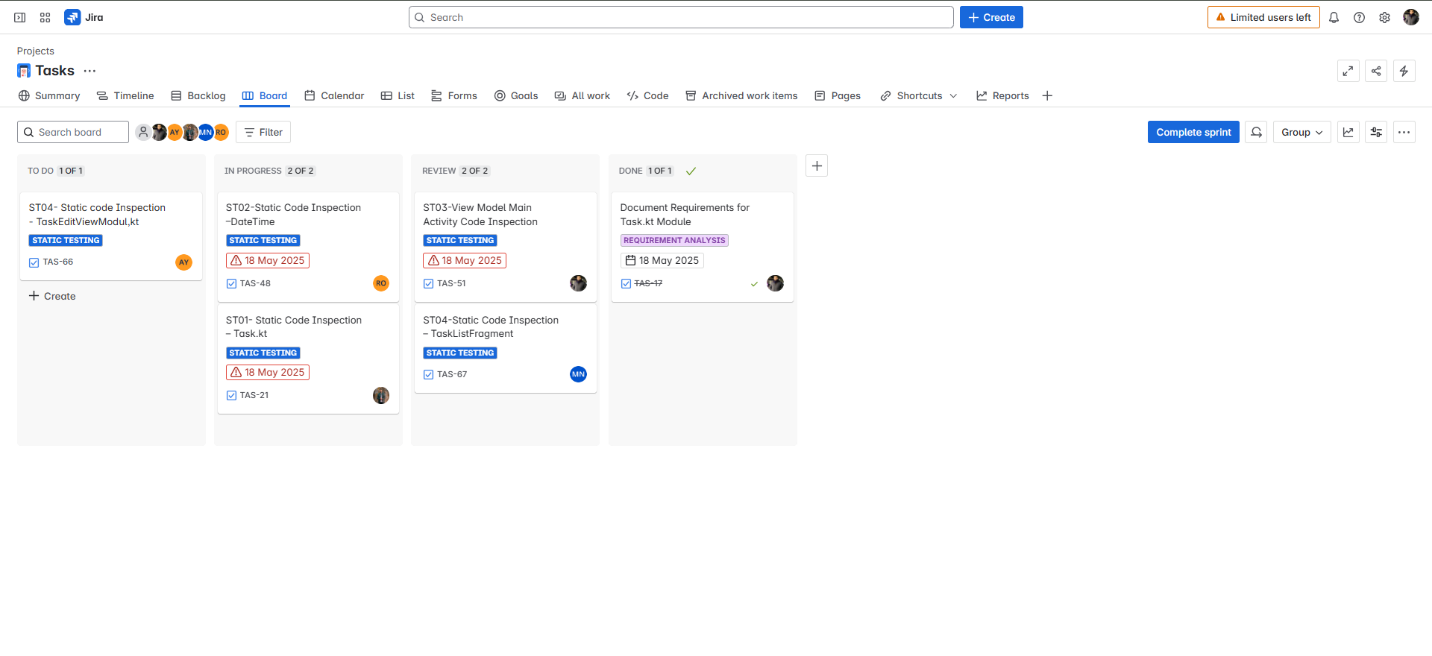


Figure 11:Static Testing and Requirement Analysis Sprint

### **3.6.3 Test Case Coverage & Jira Reports**

* Each member created at least 4 test cases (white-box or black-box).
* Each member uploaded 3 automated test tasks (tagged under "Automation").
* Jira was used to monitor which test cases were moved from "To Do" → "In Progress" → "Review" → "Done".

### **3.6.4 Integration Highlights**

* Static testing results were logged under the “Static Testing” Epic.
* Requirement tracking was linked to the “Requirement Analysis” Epic.
* Jira helped track bug fixes and regression tests after modifications.

### **3.6.5 Conclusion**

Jira enabled a structured and transparent test management process throughout the QA project. From requirement traceability to sprint planning and test execution, the tool provided full visibility and collaboration for all 5 QA members.

## **3.7 Regression Testing**

As part of our quality assurance process, we conducted thorough regression testing to ensure that recent modifications and feature enhancements did not introduce new bugs or affect existing functionalities. The regression tests focused on core components that play a vital role in the task management workflow.

The following modules were covered in our regression testing scope:

* **Task.kt**: We verified the behavior of methods such as **isCompleted(), isDeleted(), and insignificantChange(task)** to ensure task state transitions remained consistent. Additionally, synchronization-related methods like **googleTaskUpToDate(), caldavUpToDate(), and microsoftUpToDate()** were tested to confirm external service integrity.
* **TaskEditViewModel.kt**: This module underwent focused testing on critical functions like **save(), setDueDate(), and applyCalendarChanges()** to ensure task edits and calendar adjustments behaved as intended. We also examined logic branches involving **hasChanges()**, and ensured that suggested refactors for **saveAlarms(), saveTags(), and saveSubtasks()** preserved previous behavior without introducing regressions.
* **DateTime.kt**: Time manipulation and comparison are foundational to task scheduling. All constructors, arithmetic methods (e.g., **plusDays(), minusSeconds()**), and comparison methods (**isAfter(), isBefore()**) were retested. Time Zone conversions (**toUTC(), toLocal()**) were specifically verified to avoid regressions in cross-region use cases.
* **MainActivityViewModel.kt**: We ensured that the UI-driven methods such as **setFilter(), queryMenu(), and toggleCollapsed()** continued to operate as expected after refactoring. The **updateFilters()** logic was carefully verified to guarantee that drawer item rendering remained stable.
* **TaskListFragment.kt**: As this fragment controls the user interface for listing tasks, we validated the **onCreateView() and onMenuItemClick()** logic to ensure no functional degradation occurred. We also confirmed that **onActivityResult()** properly handled task return flows.

Following the completion of regression testing, we did not encounter any unexpected behaviors or failures in previously working features. All test cases passed successfully, confirming that the recent code changes were safely integrated and system stability was maintained.

1. **Summary of Activities and Improvements**

Throughout this QA project, our team executed a comprehensive and structured quality assurance process to validate the functionality, reliability, and performance of the Tasks Android application.

We began with a detailed **requirement analysis** to define the expected behavior and identify key features requiring coverage. This informed our test planning and ensured that both functional and non-functional requirements were addressed.

**Static code analysis** was conducted on core classes such as Task.kt, TaskEditViewModel.kt, DateTime.kt, and MainActivityViewModel.kt to catch logical inconsistencies early. We then designed and implemented both black-box and white-box test cases. Black-box testing helped validate user-facing behavior without internal knowledge of the code, while white-box testing ensured the correctness of internal logic and flow.

**JUnit** was used to develop and run unit tests targeting critical methods and logic. Performance testing was performed using **JMeter** to evaluate how the app handled varying load conditions and ensure its responsiveness.

To manage and track our test execution, bug reports, and progress, we used **Jira**. Test cases were organized into epics and tasks, with clear traceability to user stories and features.

As part of our **regression testing** process, we re-verified previously stable features after code updates—focusing on areas such as task handling, filtering, view model updates, and UI interactions. This ensured that recent changes did not introduce any unintended side effects. Our regression testing showed that the system remained stable and no functional regressions were found.

During the project, we also attempted to use Katalon Studio to automate UI testing. However, the tool failed to initialize properly and consistently threw environment-related errors that prevented test execution. After several troubleshooting attempts, we determined that Katalon Studio was incompatible with our current emulator/device configuration and opted to rely on JUnit and manual UI interaction testing instead.

Some challenges faced during testing included emulator inconsistencies and timing issues with reminders. These were mitigated by performing tests on real devices and adjusting logic for accurate scheduling.

**In conclusion**, our QA process effectively ensured the reliability and readiness of the Tasks application. The combined use of manual, automated, and regression testing provided confidence in both new and existing features, helping to maintain a stable and high-quality product.